Java 8

简明教程

目錄

Java 8 简明教程

作者：winterbe

原文：java8-tutorial

译者：飞龙 等

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- Github

赞助我

协议

Java 8 简明教程

> 原文：Java 8 Tutorial
>
> 译者：ImportNew.com - 黄小非
>
> 来源：Java 8简明教程

> “Java并没有没落，人们很快就会发现这一点”

欢迎阅读我编写的Java 8介绍。本教程将带领你一步一步地认识这门语言的新特性。通过简单明了的代码示例，你将会学习到如何使用默认接口方法，Lambda表达式，方法引用和重复注解。看完这篇教程后，你还将对最新推出的API有一定的了解，例如：流控制，函数式接口，map扩展和新的时间日期API等等。

允许在接口中有默认方法实现

Java 8 允许我们使用default关键字，为接口声明添加非抽象的方法实现。这个特性又被称为扩展方法。下面是我们的第一个例子：

```
interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

在接口Formula中，除了抽象方法caculate以外，还定义了一个默认方法sqrt。Formula的实现类只需要实现抽象方法caculate就可以了。默认方法sqrt可以直接使用。

```
Formula formula = new Formula() {
    @Override
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};

formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

formula对象以匿名对象的形式实现了Formula接口。代码很啰嗦：用了6行代码才实现了一个简单的计算功能：a*100开平方根。我们在下一节会看到，Java 8 还有一种更加优美的方法，能够实现包含单个函数的对象。

Lambda表达式

让我们从最简单的例子开始，来学习如何对一个string列表进行排序。我们首先使用Java 8之前的方法来实现：

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

静态工具方法Collections.sort接受一个list，和一个Comparator接口作为输入参数，Comparator的实现类可以对输入的list中的元素进行比较。通常情况下，你可以直接用创建匿名Comparator对象，并把它作为参数传递给sort方法。

除了创建匿名对象以外，Java 8 还提供了一种更简洁的方式，Lambda表达式。

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

你可以看到，这段代码就比之前的更加简短和易读。但是，它还可以更加简短：

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

只要一行代码，包含了方法体。你甚至可以连大括号对{}和return关键字都省略不要。不过这还不是最短的写法：

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Java编译器能够自动识别参数的类型，所以你就可以省略掉类型不写。让我们再深入地研究一下lambda表达式的威力吧。

函数式接口

Lambda表达式如何匹配Java的类型系统？每一个lambda都能够通过一个特定的接口，与一个给定的类型进行匹配。一个所谓的函数式接口必须要有且仅有一个抽象方法声明。每个与之对应的lambda表达式必须要与抽象方法的声明相匹配。由于默认方法不是抽象的，因此你可以在你的函数式接口里任意添加默认方法。

任意只包含一个抽象方法的接口，我们都可以用来做成lambda表达式。为了让你定义的接口满足要求，你应当在接口前加上@FunctionalInterface 标注。编译器会注意到这个标注，如果你的接口中定义了第二个抽象方法的话，编译器会抛出异常。

举例：

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}

Converter<String, Integer> converter = (from) -> Integer.valueOf
(from);
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

注意，如果你不写@FunctionalInterface 标注，程序也是正确的。

方法和构造函数引用

上面的代码实例可以通过静态方法引用，使之更加简洁：

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);   // 123
```

Java 8 允许你通过::关键字获取方法或者构造函数的的引用。上面的例子就演示了如何引用一个静态方法。而且，我们还可以对一个对象的方法进行引用：

```
class Something {
    String startsWith(String s) {
        return String.valueOf(s.charAt(0));
    }
}

Something something = new Something();
Converter<String, String> converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted);    // "J"
```

让我们看看如何使用::关键字引用构造函数。首先我们定义一个示例bean，包含不同的构造方法：

```
class Person {
    String firstName;
    String lastName;

    Person() {}

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

接下来，我们定义一个person工厂接口，用来创建新的person对象：

```
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

然后我们通过构造函数引用来把所有东西拼到一起，而不是像以前一样，通过手动实现一个工厂来这么做。

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

我们通过Person::new来创建一个Person类构造函数的引用。Java编译器会自动地选择合适的构造函数来匹配PersonFactory.create函数的签名，并选择正确的构造函数形式。

Lambda的范围

对于lambdab表达式外部的变量，其访问权限的粒度与匿名对象的方式非常类似。你能够访问局部对应的外部区域的局部final变量，以及成员变量和静态变量。

访问局部变量

我们可以访问lambda表达式外部的final局部变量：

```
final int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

但是与匿名对象不同的是，变量num并不需要一定是final。下面的代码依然是合法的：

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

然而，num在编译的时候被隐式地当做final变量来处理。下面的代码就不合法：

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
num = 3;
```

在lambda表达式内部企图改变num的值也是不允许的。

访问成员变量和静态变量

与局部变量不同，我们在lambda表达式的内部能获取到对成员变量或静态变量的读写权。这种访问行为在匿名对象里是非常典型的。

```
class Lambda4 {
    static int outerStaticNum;
    int outerNum;

    void testScopes() {
        Converter<Integer, String> stringConverter1 = (from) ->
{
            outerNum = 23;
            return String.valueOf(from);
        };

        Converter<Integer, String> stringConverter2 = (from) ->
{
            outerStaticNum = 72;
            return String.valueOf(from);
        };
    }
}
```

访问默认接口方法

还记得第一节里面formula的那个例子么？ 接口Formula定义了一个默认的方法sqrt，该方法能够访问formula所有的对象实例，包括匿名对象。这个对lambda表达式来讲则无效。

默认方法无法在lambda表达式内部被访问。因此下面的代码是无法通过编译的：

```
Formula formula = (a) -> sqrt( a * 100);
```

内置函数式接口

JDK 1.8 API中包含了很多内置的函数式接口。有些是在以前版本的Java中大家耳熟能详的，例如Comparator接口，或者Runnable接口。对这些现成的接口进行实现，可以通过@FunctionalInterface 标注来启用Lambda功能支持。

此外，Java 8 API 还提供了很多新的函数式接口，来降低程序员的工作负担。有些新的接口已经在Google Guava库中很有名了。如果你对这些库很熟的话，你甚至闭上眼睛都能够想到，这些接口在类库的实现过程中起了多么大的作用。

Predicates

Predicate是一个布尔类型的函数，该函数只有一个输入参数。Predicate接口包含了多种默认方法，用于处理复杂的逻辑动词（and, or，negate）

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");              // true
predicate.negate().test("foo");     // false

Predicate<Boolean> nonNull = Objects::nonNull;
Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

Functions

Function接口接收一个参数，并返回单一的结果。默认方法可以将多个函数串在一起（compse, andThen）

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String
::valueOf);

backToString.apply("123");     // "123"
```

Suppliers

Supplier接口产生一个给定类型的结果。与Function不同的是，Supplier没有输入参数。

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();   // new Person
```

Consumers

Consumer代表了在一个输入参数上需要进行的操作。

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

Comparators

Comparator接口在早期的Java版本中非常著名。Java 8 为这个接口添加了不同的默认方法。

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);             // > 0
comparator.reversed().compare(p1, p2);  // < 0
```

Optionals

Optional不是一个函数式接口，而是一个精巧的工具接口，用来防止NullPointerException产生。这个概念在下一节会显得很重要，所以我们在这里快速地浏览一下Optional的工作原理。

Optional是一个简单的值容器，这个值可以是null，也可以是non-null。考虑到一个方法可能会返回一个non-null的值，也可能返回一个空值。为了不直接返回null，我们在Java 8中就返回一个Optional.

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();           // true
optional.get();                 // "bam"
optional.orElse("fallback");    // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0)));
// "b"
```

Streams

java.util.Stream表示了某一种元素的序列，在这些元素上可以进行各种操作。Stream操作可以是中间操作，也可以是完结操作。完结操作会返回一个某种类型的值，而中间操作会返回流对象本身，并且你可以通过多次调用同一个流操作方法来将操作结果串起来（就像StringBuffer的append方法一样————译者注）。Stream是在一个源的基础上创建出来的，例如java.util.Collection中的list或者set（map不能作为Stream的源）。Stream操作往往可以通过顺序或者并行两种方式来执行。

我们先了解一下序列流。首先，我们通过string类型的list的形式创建示例数据：

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

Java 8中的Collections类的功能已经有所增强，你可以之直接通过调用Collections.stream()或者Collection.parallelStream()方法来创建一个流对象。下面的章节会解释这个最常用的操作。

Filter

Filter接受一个predicate接口类型的变量，并将所有流对象中的元素进行过滤。该操作是一个中间操作，因此它允许我们在返回结果的基础上再进行其他的流操作（forEach）。ForEach接受一个function接口类型的变量，用来执行对每一个元素的操作。ForEach是一个中止操作。它不返回流，所以我们不能再调用其他的流操作。

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa2", "aaa1"
```

Sorted

Sorted是一个中间操作，能够返回一个排过序的流对象的视图。流对象中的元素会默认按照自然顺序进行排序，除非你自己指定一个Comparator接口来改变排序规则。

```
stringCollection
    .stream()
    .sorted()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa1", "aaa2"
```

一定要记住，sorted只是创建一个流对象排序的视图，而不会改变原来集合中元素的顺序。原来string集合中的元素顺序是没有改变的。

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Map

map是一个对于流对象的中间操作，通过给定的方法，它能够把流对象中的每一个元素对应到另外一个对象上。下面的例子就演示了如何把每个string都转换成大写的string. 不但如此，你还可以把每一种对象映射成为其他类型。对于带泛型结果的流对象，具体的类型还要由传递给map的泛型方法来决定。

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Match

匹配操作有多种不同的类型，都是用来判断某一种规则是否与流对象相互吻合的。所有的匹配操作都是终结操作，只返回一个boolean类型的结果。

```
boolean anyStartsWithA =
    stringCollection
        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);      // true

boolean allStartsWithA =
    stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);      // false

boolean noneStartsWithZ =
    stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);      // true
```

Count

Count是一个终结操作，它的作用是返回一个数值，用来标识当前流对象中包含的元素数量。

```
long startsWithB =
    stringCollection
        .stream()
        .filter((s) -> s.startsWith("b"))
        .count();

System.out.println(startsWithB);    // 3
```

Reduce

该操作是一个终结操作，它能够通过某一个方法，对元素进行削减操作。该操作的结果会放在一个Optional变量里返回。

```
Optional<String> reduced =
    stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

Parallel Streams

像上面所说的，流操作可以是顺序的，也可以是并行的。顺序操作通过单线程执行，而并行操作则通过多线程执行。

下面的例子就演示了如何使用并行流进行操作来提高运行效率，代码非常简单。

首先我们创建一个大的list，里面的元素都是唯一的：

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
    UUID uuid = UUID.randomUUID();
    values.add(uuid.toString());
}
```

现在，我们测量一下对这个集合进行排序所使用的时间。

顺序排序

```
long t0 = System.nanoTime();

long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("sequential sort took: %d ms",
millis));

// sequential sort took: 899 ms
```

并行排序

```
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("parallel sort took: %d ms", mi
llis));

// parallel sort took: 472 ms
```

如你所见，所有的代码段几乎都相同，唯一的不同就是把stream()改成了parallelStream(), 结果并行排序快了50%。

Map

正如前面已经提到的那样，map是不支持流操作的。而更新后的map现在则支持多种实用的新方法，来完成常规的任务。

```
Map<Integer, String> map = new HashMap<>();

for (int i = 0; i < 10; i++) {
    map.putIfAbsent(i, "val" + i);
}

map.forEach((id, val) -> System.out.println(val));
```

上面的代码风格是完全自解释的：putIfAbsent避免我们将null写入；forEach接受一个消费者对象，从而将操作实施到每一个map中的值上。

下面的这个例子展示了如何使用函数来计算map的编码

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3);             // val33

map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9);     // false

map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23);    // true

map.computeIfAbsent(3, num -> "bam");
map.get(3);             // val33
```

接下来，我们将学习，当给定一个key值时，如何把一个实例从对应的key中移除：

```
map.remove(3, "val3");
map.get(3);             // val33

map.remove(3, "val33");
map.get(3);             // null
```

另一个有用的方法：

```
map.getOrDefault(42, "not found");  // not found
```

将map中的实例合并也是非常容易的：

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue)
);
map.get(9);             // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValu
e));
map.get(9);             // val9concat
```

合并操作先看map中是否没有特定的key/value存在，如果是，则把key/value存入map，否则merging函数就会被调用，对现有的数值进行修改。

时间日期API

Java 8 包含了全新的时间日期API，这些功能都放在了java.time包下。新的时间日期API是基于Joda-Time库开发的，但是也不尽相同。下面的例子就涵盖了大多数新的API的重要部分。

Clock

Clock提供了对当前时间和日期的访问功能。Clock是对当前时区敏感的，并可用于替代System.currentTimeMillis()方法来获取当前的毫秒时间。当前时间线上的时刻可以用Instance类来表示。Instance也能够用于创建原先的java.util.Date对象。

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();

Instant instant = clock.instant();
Date legacyDate = Date.from(instant);   // legacy java.util.Date
```

Timezones

时区类可以用一个ZoneId来表示。时区类的对象可以通过静态工厂方法方便地获取。时区类还定义了一个偏移量，用来在当前时刻或某时间与目标时区时间之间进行转换。

```
System.out.println(ZoneId.getAvailableZoneIds());
// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

LocalTime

本地时间类表示一个没有指定时区的时间，例如，10 p.m.或者17：30:15，下面的例子会用上面的例子定义的时区创建两个本地时间对象。然后我们会比较两个时间，并计算它们之间的小时和分钟的不同。

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween);       // -3
System.out.println(minutesBetween);     // -239
```

LocalTime是由多个工厂方法组成，其目的是为了简化对时间对象实例的创建和操作，包括对时间字符串进行解析的操作。

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);       // 23:59:59

DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedTime(FormatStyle.SHORT)
        .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);   // 13:37
```

LocalDate

本地时间表示了一个独一无二的时间，例如：2014-03-11。这个时间是不可变的，与LocalTime是同源的。下面的例子演示了如何通过加减日，月，年等指标来计算新的日期。记住，每一次操作都会返回一个新的时间对象。

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek);    // FRIDAY<span style="font-fam
ily: Georgia, 'Times New Roman', 'Bitstream Charter', Times, ser
if; font-size: 13px; line-height: 19px;">Parsing a LocalDate fro
m a string is just as simple as parsing a LocalTime:</span>
```

解析字符串并形成LocalDate对象，这个操作和解析LocalTime一样简单。

```
DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedDate(FormatStyle.MEDIUM)
        .withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
System.out.println(xmas);   // 2014-12-24
```

LocalDateTime

LocalDateTime表示的是日期-时间。它将刚才介绍的日期对象和时间对象结合起来，形成了一个对象实例。LocalDateTime是不可变的，与LocalTime和LocalDate的工作原理相同。我们可以通过调用方法来获取日期时间对象中特定的数据域。

```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER,
31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek);      // WEDNESDAY

Month month = sylvester.getMonth();
System.out.println(month);          // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay);    // 1439
```

如果再加上的时区信息，LocalDateTime能够被转换成Instance实例。Instance能够被转换成以前的java.util.Date对象。

```
Instant instant = sylvester
        .atZone(ZoneId.systemDefault())
        .toInstant();

Date legacyDate = Date.from(instant);
System.out.println(legacyDate);     // Wed Dec 31 23:59:59 CET 2
014
```

格式化日期-时间对象就和格式化日期对象或者时间对象一样。除了使用预定义的格式以外，我们还可以创建自定义的格式化对象，然后匹配我们自定义的格式。

```
DateTimeFormatter formatter =
    DateTimeFormatter
        .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13"
, formatter);
String string = formatter.format(parsed);
System.out.println(string);     // Nov 03, 2014 - 07:13
```

不同于java.text.NumberFormat，新的DateTimeFormatter类是不可变的，也是线程安全的。

更多的细节，请看这里

Annotations

Java 8中的注解是可重复的。让我们直接深入看看例子，弄明白它是什么意思。

首先，我们定义一个包装注解，它包括了一个实际注解的数组

```
@interface Hints {
    Hint[] value();
}

@Repeatable(Hints.class)
@interface Hint {
    String value();
}
```

只要在前面加上注解名：@Repeatable，Java 8 允许我们对同一类型使用多重注解，

变体1：使用注解容器（老方法）

```
@Hints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

变体2：使用可重复注解（新方法）

```
@Hint("hint1")
@Hint("hint2")
class Person {}
```

使用变体2，Java编译器能够在内部自动对@Hint进行设置。这对于通过反射来读取注解信息来说，是非常重要的。

```
Hint hint = Person.class.getAnnotation(Hint.class);
System.out.println(hint);                   // null

Hints hints1 = Person.class.getAnnotation(Hints.class);
System.out.println(hints1.value().length);  // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length);          // 2
```

尽管我们绝对不会在Person类上声明@Hints注解，但是它的信息仍然可以通过getAnnotation(Hints.class)来读取。并且，getAnnotationsByType方法会更方便，因为它赋予了所有@Hints注解标注的方法直接的访问权限。

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

先到这里

我的Java 8编程指南就到此告一段落。当然，还有很多内容需要进一步研究和说明。这就需要靠读者您来对JDK 8进行探究了，例如：Arrays.parallelSort, StampedLock和CompletableFuture等等 ———— 我这里只是举几个例子而已。

我希望这个博文能够对您有所帮助，也希望您阅读愉快。完整的教程源代码放在了GitHub上。您可以尽情地fork，并请通过Twitter告诉我您的反馈。

Java 8 数据流教程

> 原文：Java 8 Stream Tutorial
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

这个示例驱动的教程是Java8数据流（Stream）的深入总结。当我第一次看到 `Stream` API时，我非常疑惑，因为它听起来和Java IO的 `InputStream` 和 `OutputStream` 一样。但是Java8的数据流是完全不同的东西。数据流是单体（Monad），并且在Java8函数式编程中起到重要作用。

> 在函数式编程中，单体是一个结构，表示定义为步骤序列的计算。单体结构的类型定义了它对链式操作，或具有相同类型的嵌套函数的含义。

这个教程教给你如何使用Java8数据流，以及如何使用不同种类的可用的数据流操作。你将会学到处理次序以及流操作的次序如何影响运行时效率。这个教程也会详细讲解更加强大的流操作， `reduce` 、 `collect` 和 `flatMap` 。最后，这个教程会深入探讨并行流。

如果你还不熟悉Java8的lambda表达式，函数式接口和方法引用，你可能需要在开始这一章之前，首先阅读我的Java8教程。

更新 - 我现在正在编写用于浏览器的Java8数据流API的JavaScript实现。如果你对此感兴趣，请在Github上访问Stream.js。非常期待你的反馈。

数据流如何工作

数据流表示元素的序列，并支持不同种类的操作来执行元素上的计算：

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

// C1
// C2
```

数据流操作要么是衔接操作，要么是终止操作。衔接操作返回数据流，所以我们可以把多个衔接操作不使用分号来链接到一起。终止操作无返回值，或者返回一个不是流的结果。在上面的例子中， `filter` 、 `map` 和 `sorted` 都是衔接操作，而 `forEach` 是终止操作。列表上的所有流式操作请见数据流的Javadoc。你在上面例子中看到的这种数据流的链式操作也叫作操作流水线。

多数数据流操作都接受一些lambda表达式参数，函数式接口用来指定操作的具体行为。这些操作的大多数必须是无干扰而且是无状态的。它们是什么意思呢？

当一个函数不修改数据流的底层数据源，它就是无干扰的。例如，在上面的例子中，没有任何lambda表达式通过添加或删除集合元素修改 `myList` 。

当一个函数的操作的执行是确定性的，它就是无状态的。例如，在上面的例子中，没有任何lambda表达式依赖于外部作用域中任何在操作过程中可变的变量或状态。

数据流的不同类型

数据流可以从多种数据源创建，尤其是集合。 `List` 和 `Set` 支持新方法 `stream()` 和 `parallelStream()` ，来创建串行流或并行流。并行流能够在多个线程上执行操作，它们会在之后的章节中讲到。我们现在来看看串行流：

```
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

在对象列表上调用 `stream()` 方法会返回一个通常的对象流。但是我们不需要创建一个集合来创建数据流，就像下面那样：

```
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

只要使用 `Stream.of()` ，就可以从一系列对象引用中创建数据流。

除了普通的对象数据流，Java8还自带了特殊种类的流，用于处理基本数据类型 `int` 、 `long` 和 `double` 。你可能已经猜到了它是 `IntStream` 、 `LongStream` 和 `DoubleStream` 。

`IntStream` 可以使用 `IntStream.range()` 替换通常的 `for` 循环：

```
IntStream.range(1, 4)
    .forEach(System.out::println);

// 1
// 2
// 3
```

所有这些基本数据流都像通常的对象数据流一样，但有一些不同。基本的数据流使用特殊的lambda表达式，例如，`IntFunction` 而不是 `Function`，`IntPredicate` 而不是 `Predicate`。而且基本数据流支持额外的聚合终止操作 `sum()` 和 `average()`：

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println);  // 5.0
```

有时需要将通常的对象数据流转换为基本数据流，或者相反。出于这种目的，对象数据流支持特殊的映射操作 `mapToInt()`、`mapToLong()` 和 `mapToDouble()`：

```
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

基本数据流可以通过 `mapToObj()` 转换为对象数据流：

```
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

下面是组合示例：浮点数据流首先映射为整数数据流，之后映射为字符串的对象数据流：

```
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

处理顺序

既然我们已经了解了如何创建并使用不同种类的数据流，让我们深入了解数据流操作在背后如何执行吧。

衔接操作的一个重要特性就是延迟性。观察下面没有终止操作的例子：

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

执行这段代码时，不向控制台打印任何东西。这是因为衔接操作只在终止操作调用时被执行。

让我们通过添加终止操作 `forEach` 来扩展这个例子：

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

执行这段代码会得到如下输出：

```
filter:  d2
forEach: d2
filter:  a2
forEach: a2
filter:  b1
forEach: b1
filter:  b3
forEach: b3
filter:  c
forEach: c
```

结果的顺序可能出人意料。原始的方法会在数据流的所有元素上，一个接一个地水平执行所有操作。但是每个元素在调用链上垂直移动。第一个字符串 `"d2"` 首先经过 `filter` 然后是 `forEach` ，执行完后才开始处理第二个字符串 `"a2"` 。

这种行为可以减少每个元素上所执行的实际操作数量，就像我们在下个例子中看到的那样：

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });

// map:      d2
// anyMatch: D2
// map:      a2
// anyMatch: A2
```

只要提供的数据元素满足了谓词， `anyMatch` 操作就会返回 `true` 。对于第二个传递 `"A2"` 的元素，它的结果为真。由于数据流的链式调用是垂直执行的， `map` 这里只需要执行两次。所以 `map` 会执行尽可能少的次数，而不是把所有元素都映射一遍。

为什么顺序如此重要

下面的例子由两个衔接操作 `map` 和 `filter` ，以及一个终止操作 `forEach` 组成。让我们再来看看这些操作如何执行：

```java
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));

// map:     d2
// filter:  D2
// map:     a2
// filter:  A2
// forEach: A2
// map:     b1
// filter:  B1
// map:     b3
// filter:  B3
// map:     c
// filter:  C
```

就像你可能猜到的那样，`map` 和 `filter` 会对底层集合的每个字符串调用五次，而 `forEach` 只会调用一次。

如果我们调整操作顺序，将 `filter` 移动到调用链的顶端，就可以极大减少操作的执行次数:

```java
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// map:     a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
```

现在，`map`只会调用一次，所以操作流水线对于更多的输入元素会执行更快。在整合复杂的方法链时，要记住这一点。

让我们通过添加额外的方法`sorted`来扩展上面的例子：

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

排序是一类特殊的衔接操作。它是有状态的操作，因为你需要在处理中保存状态来对集合中的元素排序。

执行这个例子会得到如下输入：

```
sort:    a2; d2
sort:    b1; a2
sort:    b1; d2
sort:    b1; a2
sort:    b3; b1
sort:    b3; d2
sort:    c; b3
sort:    c; d2
filter:  a2
map:     a2
forEach: A2
filter:  b1
filter:  b3
filter:  c
filter:  d2
```

首先，排序操作在整个输入集合上执行。也就是说，`sorted`以水平方式执行。所以这里`sorted`对输入集合中每个元素的多种组合调用了八次。

我们同样可以通过重排调用链来优化性能：

```java
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// filter:  b1
// filter:  b3
// filter:  c
// map:     a2
// forEach: A2
```

这个例子中 `sorted` 永远不会调用，因为 `filter` 把输入集合减少至只有一个元素。所以对于更大的输入集合会极大提升性能。

复用数据流

Java8的数据流不能被复用。一旦你调用了任何终止操作，数据流就关闭了：

```java
Stream<String> stream =
    Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

在相同数据流上，在 `anyMatch` 之后调用 `noneMatch` 会产生下面的异常：

```
java.lang.IllegalStateException: stream has already been operate
d upon or closed
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeli
ne.java:229)
    at java.util.stream.ReferencePipeline.noneMatch(ReferencePip
eline.java:459)
    at com.winterbe.java8.Streams5.test7(Streams5.java:38)
    at com.winterbe.java8.Streams5.main(Streams5.java:28)
```

要克服这个限制，我们需要为每个我们想要执行的终止操作创建新的数据流调用链。例如，我们创建一个数据流供应器，来构建新的数据流，并且设置好所有衔接操作：

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
            .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

每次对 `get()` 的调用都构造了一个新的数据流，我们将其保存来调用终止操作。

高级操作

数据流执行大量的不同操作。我们已经了解了一些最重要的操作，例如 `filter` 和 `map` 。我将它们留给你来探索所有其他的可用操作（请见数据流的Javadoc）。下面让我们深入了解一些更复杂的操作： `collect` 、 `flatMap` 和 `reduce` 。

这一节的大部分代码示例使用下面的 `Person` 列表来演示：

```java
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name;
    }
}

List<Person> persons =
    Arrays.asList(
        new Person("Max", 18),
        new Person("Peter", 23),
        new Person("Pamela", 23),
        new Person("David", 12));
```

`collect`

`collect` 是非常有用的终止操作，将流中的元素存放在不同类型的结果中，例如 `List` 、 `Set` 或者 `Map` 。 `collect` 接受收集器（Collector），它由四个不同的操作组成：供应器（supplier）、累加器（accumulator）、组合器（combiner）和终止器（finisher）。这在开始听起来十分复杂，但是Java8通过内置的 `Collectors` 类支持多种内置的收集器。所以对于大部分常见操作，你并不需要自己实现收集器。

让我们以一个非常常见的用例来开始：

```java
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());

System.out.println(filtered);    // [Peter, Pamela]
```

就像你看到的那样，它非常简单，只是从流的元素中构造了一个列表。如果需要以 `Set` 来替代 `List` ，只需要使用 `Collectors.toSet()` 就好了。

下面的例子按照年龄对所有人进行分组：

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age,
p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

收集器十分灵活。你也可以在流的元素上执行聚合，例如，计算所有人的平均年龄：

```
Double averageAge = persons
    .stream()
    .collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge);     // 19.0
```

如果你对更多统计学方法感兴趣，概要收集器返回一个特殊的内置概要统计对象，所以我们可以简单计算最小年龄、最大年龄、算术平均年龄、总和和数量。

```
IntSummaryStatistics ageSummary =
    persons
        .stream()
        .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.0000
00, max=23}
```

下面的例子将所有人连接为一个字符串：

```
String phrase = persons
    .stream()
    .filter(p -> p.age >= 18)
    .map(p -> p.name)
    .collect(Collectors.joining(" and ", "In Germany ", " are of
 legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

连接收集器接受分隔符，以及可选的前缀和后缀。

为了将数据流中的元素转换为映射，我们需要指定键和值如何被映射。要记住键必须是唯一的，否则会抛出 `IllegalStateException` 异常。你可以选择传递一个合并函数作为额外的参数来避免这个异常。

既然我们知道了一些最强大的内置收集器，让我们来尝试构建自己的特殊收集器吧。我们希望将流中的所有人转换为一个字符串，包含所有大写的名称，并以 `|` 分割。为了完成它，我们通过 `Collector.of()` 创建了一个新的收集器。我们需要传递一个收集器的四个组成部分：供应器、累加器、组合器和终止器。

```
Collector<Person, StringJoiner, String> personNameCollector =
    Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
    .stream()
    .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID
```

由于Java中的字符串是不可变的，我们需要一个助手类 `StringJointer` 。让收集器构造我们的字符串。供应器最开始使用相应的分隔符构造了这样一个 `StringJointer` 。累加器用于将每个人的大写名称加到 `StringJointer` 中。组合器知道如何把两个 `StringJointer` 合并为一个。最后一步，终结器从 `StringJointer` 构造出预期的字符串。

`flatMap`

我们已经了解了如何通过使用 `map` 操作，将流中的对象转换为另一种类型。 `map` 有时十分受限，因为每个对象只能映射为一个其它对象。但如何我希望将一个对象转换为多个或零个其他对象呢？ `flatMap` 这时就会派上用场。

`flatMap` 将流中的每个元素，转换为其它对象的流。所以每个对象会被转换为零个、一个或多个其它对象，以流的形式返回。这些流的内容之后会放进 `flatMap` 所返回的流中。

在我们了解 `flatMap` 如何使用之前，我们需要相应的类型体系：

```
class Foo {
    String name;
    List<Bar> bars = new ArrayList<>();

    Foo(String name) {
        this.name = name;
    }
}

class Bar {
    String name;

    Bar(String name) {
        this.name = name;
    }
}
```

下面，我们使用我们自己的关于流的知识来实例化一些对象：

```
List<Foo> foos = new ArrayList<>();

// create foos
IntStream
    .range(1, 4)
    .forEach(i -> foos.add(new Foo("Foo" + i)));

// create bars
foos.forEach(f ->
    IntStream
        .range(1, 4)
        .forEach(i -> f.bars.add(new Bar("Bar" + i + " <- " + f.
name))));
```

现在我们拥有了含有三个 `foo` 的列表，每个都含有三个 `bar` 。

`flatMap` 接受返回对象流的函数。所以为了处理每个 `foo` 上的 `bar` 对象，我们需要传递相应的函数：

```
foos.stream()
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));

// Bar1 <- Foo1
// Bar2 <- Foo1
// Bar3 <- Foo1
// Bar1 <- Foo2
// Bar2 <- Foo2
// Bar3 <- Foo2
// Bar1 <- Foo3
// Bar2 <- Foo3
// Bar3 <- Foo3
```

像你看到的那样，我们成功地将含有三个 `foo` 对象中的流转换为含有九个 `bar` 对象的流。

最后，上面的代码示例可以简化为流式操作的单一流水线：

```
IntStream.range(1, 4)
    .mapToObj(i -> new Foo("Foo" + i))
    .peek(f -> IntStream.range(1, 4)
        .mapToObj(i -> new Bar("Bar" + i + " <- " + f.name))
        .forEach(f.bars::add))
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));
```

`flatMap` 也可用于Java8引入的 `Optional` 类。 `Optional` 的 `flatMap` 操作返回一个 `Optional` 或其他类型的对象。所以它可以用于避免烦人的 `null` 检查。

考虑像这样更复杂的层次结构：

```
class Outer {
    Nested nested;
}

class Nested {
    Inner inner;
}

class Inner {
    String foo;
}
```

为了处理外层示例上的内层字符串 `foo` ，你需要添加多个 `null` 检查来避免潜在的 `NullPointerException` ：

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner
!= null) {
    System.out.println(outer.nested.inner.foo);
}
```

可以使用 `Optional` 的 `flatMap` 操作来完成相同的行为：

```
Optional.of(new Outer())
    .flatMap(o -> Optional.ofNullable(o.nested))
    .flatMap(n -> Optional.ofNullable(n.inner))
    .flatMap(i -> Optional.ofNullable(i.foo))
    .ifPresent(System.out::println);
```

如果存在的话，每个 `flatMap` 的调用都会返回预期对象的 `Optional` 包装，否则为 `null` 的 `Optional` 包装。

`reduce`

归约操作将所有流中的元素组合为单一结果。Java8支持三种不同类型的 `reduce` 方法。第一种将流中的元素归约为流中的一个元素。让我们看看我们如何使用这个方法来计算出最老的人：

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);    // Pamela
```

`reduce` 方法接受 `BinaryOperator` 积累函数。它实际上是两个操作数类型相同的 `BiFunction` 。 `BiFunction` 就像是 `Function` ，但是接受两个参数。示例中的函数比较两个人的年龄，来返回年龄较大的人。

第二个 `reduce` 方法接受一个初始值，和一个 `BinaryOperator` 累加器。这个方法可以用于从流中的其它 `Person` 对象中构造带有聚合后名称和年龄的新 `Person` 对象。

```
Person result =
    persons
        .stream()
        .reduce(new Person("", 0), (p1, p2) -> {
            p1.age += p2.age;
            p1.name += p2.name;
            return p1;
        });

System.out.format("name=%s; age=%s", result.name, result.age);
// name=MaxPeterPamelaDavid; age=76
```

第三个 `reduce` 对象接受三个参数：初始值， `BiFunction` 累加器和 `BinaryOperator` 类型的组合器函数。由于初始值的类型不一定为 `Person` ，我们可以使用这个归约函数来计算所有人的年龄总和。：

```
Integer ageSum = persons
    .stream()
    .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 +
sum2);

System.out.println(ageSum);  // 76
```

你可以看到结果是76。但是背后发生了什么？让我们通过添加一些调试输出来扩展上面的代码：

```
Integer ageSum = persons
    .stream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s\n"
, sum, p);
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", su
m1, sum2);
            return sum1 + sum2;
        });

// accumulator: sum=0; person=Max
// accumulator: sum=18; person=Peter
// accumulator: sum=41; person=Pamela
// accumulator: sum=64; person=David
```

你可以看到，累加器函数做了所有工作。它首先使用初始值 `0` 和第一个人Max来调用累加器。接下来的三步中 `sum` 会持续增加，直到76。

等一下。好像组合器从来没有调用过？以并行方式执行相同的流会揭开这个秘密：

```
Integer ageSum = persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s\n"
, sum, p);
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", su
m1, sum2);
            return sum1 + sum2;
        });

// accumulator: sum=0; person=Pamela
// accumulator: sum=0; person=David
// accumulator: sum=0; person=Max
// accumulator: sum=0; person=Peter
// combiner: sum1=18; sum2=23
// combiner: sum1=23; sum2=12
// combiner: sum1=41; sum2=35
```

这个流的并行执行行为会完全不同。现在实际上调用了组合器。由于累加器被并行调用，组合器需要用于计算部分累加值的总和。

下一节我们会深入了解并行流。

并行流

流可以并行执行，在大量输入元素上可以提升运行时的性能。并行流使用公共的 `ForkJoinPool` ，由 `ForkJoinPool.commonPool()` 方法提供。底层线程池的大小最大为五个线程 -- 取决于CPU的物理核数。

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism());    // 3
```

在我的机器上，公共池默认初始化为3。这个值可以通过设置下列JVM参数来增减：

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

集合支持 `parallelStream()` 方法来创建元素的并行流。或者你可以在已存在的数据流上调用衔接方法 `parallel()` ，将串行流转换为并行流。

为了描述并行流的执行行为，下面的例子向 `sout` 打印了当前线程的信息。

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

通过分析调试输出，我们可以对哪个线程用于执行流式操作拥有更深入的理解：

```
filter:  b1 [main]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  c2 [ForkJoinPool.commonPool-worker-3]
map:     c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map:     b1 [main]
forEach: B1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-3]
map:     a1 [ForkJoinPool.commonPool-worker-3]
forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

就像你看到的那样，并行流使用了所有公共的 `ForkJoinPool` 中的可用线程来执行流式操作。在连续的运行中输出可能有所不同，因为所使用的特定线程是非特定的。

让我们通过添加额外的流式操作 `sort` 来扩展这个示例：

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
        System.out.format("sort: %s <> %s [%s]\n",
            s1, s2, Thread.currentThread().getName());
        return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

结果起初可能比较奇怪：

```
filter:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  b1 [main]
map:     b1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-2]
map:     a1 [ForkJoinPool.commonPool-worker-2]
map:     c2 [ForkJoinPool.commonPool-worker-3]
sort:    A2 <> A1 [main]
sort:    B1 <> A2 [main]
sort:    C2 <> B1 [main]
sort:    C1 <> C2 [main]
sort:    C1 <> B1 [main]
sort:    C1 <> C2 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-1]
```

`sort` 看起来只在主线程上串行执行。实际上，并行流上的 `sort` 在背后使用了Java8中新的方法 `Arrays.parallelSort()` 。如javadoc所说，这个方法会参照数据长度来决定以串行或并行来执行。

> 如果指定数据的长度小于最小粒度，它使用相应的`Arrays.sort`方法来排序。

返回上一节中`reduce`的例子。我们已经发现了组合器函数只在并行流中调用，而不在串行流中调用。让我们来观察实际上涉及到哪个线程：

```
List<Person> persons = Arrays.asList(
    new Person("Max", 18),
    new Person("Peter", 23),
    new Person("Pamela", 23),
    new Person("David", 12));

persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s [%
s]\n",
                sum, p, Thread.currentThread().getName());
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s [%s]\n"
,
                sum1, sum2, Thread.currentThread().getName());
            return sum1 + sum2;
        });
```

控制台的输出表明，累加器和组合器都在所有可用的线程上并行执行：

```
accumulator: sum=0; person=Pamela; [main]
accumulator: sum=0; person=Max;    [ForkJoinPool.commonPool-work
er-3]
accumulator: sum=0; person=David;  [ForkJoinPool.commonPool-work
er-2]
accumulator: sum=0; person=Peter;  [ForkJoinPool.commonPool-work
er-1]
combiner:    sum1=18; sum2=23;     [ForkJoinPool.commonPool-work
er-1]
combiner:    sum1=23; sum2=12;     [ForkJoinPool.commonPool-work
er-2]
combiner:    sum1=41; sum2=35;     [ForkJoinPool.commonPool-work
er-2]
```

总之，并行流对拥有大量输入元素的数据流具有极大的性能提升。但是要记住一些并行流的操作，例如`reduce`和`collect`需要额外的计算（组合操作），这在串行执行时并不需要。

此外我们已经了解，所有并行流操作都共享相同的JVM相关的公共 `ForkJoinPool` 。所以你可能需要避免实现又慢又卡的流式操作，因为它可能会拖慢你应用中严重依赖并行流的其它部分。

到此为止

我的Java8数据流编程教程就此告一段落。如果你对深入了解Java8数据流感兴趣，我向你推荐数据流的Javadoc。如果你希望学到更多底层机制，你可能需要阅读Martin Fowler关于集合流水线的文章。

如果你对JavaScript也感兴趣，你可能希望看一看Stream.js -- 一个Java8数据流API的JavaScript实现。你也可能希望阅读我的Java8简明教程，和我的Java8Nashron教程。

我希望你会喜欢这篇文章。如果你有任何的问题都可以在下面评论或者通过 Twitter 给我回复。

祝编程愉快！

Java 8 Nashorn 教程

> 原文：Java 8 Nashorn Tutorial
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

这个教程中，你会通过简单易懂的代码示例，来了解Nashorn JavaScript引擎。Nashorn JavaScript引擎是Java SE 8 的一部分，并且和其它独立的引擎例如Google V8（用于Google Chrome和Node.js的引擎）互相竞争。Nashorn通过在JVM上，以原生方式运行动态的JavaScript代码来扩展Java的功能。

在接下来的15分钟内，你会学到如何在JVM上在运行时动态执行JavaScript。我会使用小段代码示例来演示最新的Nashron语言特性。你会学到如何在Java代码中调用JavaScript函数，或者相反。最后你会准备好将动态脚本集成到你的Java日常业务中。



更新 - 我现在正在编写用于浏览器的Java8数据流API的JavaScript实现。如果你对此感兴趣，请在Github上访问Stream.js。非常期待你的反馈。

使用 Nashron

Nashorn JavaScript引擎可以在Java代码中编程调用，也可以通过命令行工具 `jjs` 使用，它在 `$JAVA_HOME/bin` 中。如果打算使用 `jjs` ，你可能希望设置符号链接来简化访问：

```
$ cd /usr/bin
$ ln -s $JAVA_HOME/bin/jjs jjs
$ jjs
jjs> print('Hello World');
```

这个教程专注于在Java代码中调用Nashron，所以让我们先跳过 `jjs` 。Java代码中简单的HelloWorld如下所示：

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName(
"nashorn");
engine.eval("print('Hello World!');");
```

为了在Java中执行JavaScript，你首先要通过 `javax.script` 包创建脚本引擎。这个包已经在Rhino（来源于Mozilla、Java中的遗留JS引擎）中使用了。

JavaScript代码既可以通过传递JavaScript代码字符串，也可以传递指向你的JS脚本文件的 `FileReader` 来执行：

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName(
"nashorn");
engine.eval(new FileReader("script.js"));
```

Nashorn JavaScript基于ECMAScript 5.1，但是它的后续版本会对ES6提供支持：

> Nashorn的当前策略遵循ECMAScript规范。当我们在JDK8中发布它时，它将基于ECMAScript 5.1。Nashorn未来的主要发布基于ECMAScript 6。

Nashorn定义了大量对ECMAScript标准的语言和API扩展。但是首先让我们看一看Java和JavaScript代码如何交互。

在Java中调用JavaScript函数

Nashorn 支持从Java代码中直接调用定义在脚本文件中的JavaScript函数。你可以将Java对象传递为函数参数，并且从函数返回数据来调用Java方法。

下面的JavaScript函数稍后会在Java端调用：

```
var fun1 = function(name) {
    print('Hi there from Javascript, ' + name);
    return "greetings from javascript";
};

var fun2 = function (object) {
    print("JS Class Definition: " + Object.prototype.toString.ca
ll(object));
};
```

为了调用函数，你首先需要将脚本引擎转换为 `Invocable` 。 `Invocable` 接口由 `NashornScriptEngine` 实现，并且定义了 `invokeFunction` 方法来调用指定名称的JavaScript函数。

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName(
"nashorn");
engine.eval(new FileReader("script.js"));

Invocable invocable = (Invocable) engine;

Object result = invocable.invokeFunction("fun1", "Peter Parker")
;
System.out.println(result);
System.out.println(result.getClass());

// Hi there from Javascript, Peter Parker
// greetings from javascript
// class java.lang.String
```

执行这段代码会在控制台产生三行结果。调用函数 `print` 将结果输出到 `System.out` ，所以我们会首先看到JavaScript输出。

现在让我们通过传入任意Java对象来调用第二个函数：

```
invocable.invokeFunction("fun2", new Date());
// [object java.util.Date]

invocable.invokeFunction("fun2", LocalDateTime.now());
// [object java.time.LocalDateTime]

invocable.invokeFunction("fun2", new Person());
// [object com.winterbe.java8.Person]
```

Java对象在传入时不会在JavaScript端损失任何类型信息。由于脚本在JVM上原生运行，我们可以在Nashron上使用Java API或外部库的全部功能。

在JavaScript中调用Java方法

在JavaScript中调用Java方法十分容易。我们首先需要定义一个Java静态方法。

```
static String fun1(String name) {
    System.out.format("Hi there from Java, %s", name);
    return "greetings from java";
}
```

Java类可以通过 `Java.type` API扩展在JavaScript中引用。它就和Java代码中的 `import` 类似。只要定义了Java类型，我们就可以自然地调用静态方法 `fun1()` ，然后像 `sout` 打印信息。由于方法是静态的，我们不需要首先创建实例。

```
var MyJavaClass = Java.type('my.package.MyJavaClass');

var result = MyJavaClass.fun1('John Doe');
print(result);

// Hi there from Java, John Doe
// greetings from java
```

在使用JavaScript原生类型调用Java方法时，Nashorn 如何处理类型转换？让我们通过简单的例子来弄清楚。

下面的Java方法简单打印了方法参数的实际类型：

```
static void fun2(Object object) {
    System.out.println(object.getClass());
}
```

为了理解背后如何处理类型转换，我们使用不同的JavaScript类型来调用这个方法：

```
MyJavaClass.fun2(123);
// class java.lang.Integer

MyJavaClass.fun2(49.99);
// class java.lang.Double

MyJavaClass.fun2(true);
// class java.lang.Boolean

MyJavaClass.fun2("hi there")
// class java.lang.String

MyJavaClass.fun2(new Number(23));
// class jdk.nashorn.internal.objects.NativeNumber

MyJavaClass.fun2(new Date());
// class jdk.nashorn.internal.objects.NativeDate

MyJavaClass.fun2(new RegExp());
// class jdk.nashorn.internal.objects.NativeRegExp

MyJavaClass.fun2({foo: 'bar'});
// class jdk.nashorn.internal.scripts.J04
```

JavaScript原始类型转换为合适的Java包装类，而JavaScript原生对象会使用内部的适配器类来表示。要记住 `jdk.nashorn.internal` 中的类可能会有所变化，所以不应该在客户端面向这些类来编程。

> 任何标记为“内部”的东西都可能会从你那里发生改变。

`ScriptObjectMirror`

在向Java传递原生JavaScript对象时，你可以使用 `ScriptObjectMirror` 类，它实际上是底层JavaScript对象的Java表示。 `ScriptObjectMirror` 实现了 `Map` 接口，位于 `jdk.nashorn.api` 中。这个包中的类可以用于客户端代码。

下面的例子将参数类型从 `Object` 改为 `ScriptObjectMirror` ，所以我们可以从传入的JavaScript对象中获得一些信息。

```
static void fun3(ScriptObjectMirror mirror) {
    System.out.println(mirror.getClassName() + ": " +
        Arrays.toString(mirror.getOwnKeys(true)));
}
```

当向这个方法传递对象（哈希表）时，在Java端可以访问其属性：

```
MyJavaClass.fun3({
    foo: 'bar',
    bar: 'foo'
});

// Object: [foo, bar]
```

我们也可以在Java中调用JavaScript的成员函数。让我们首先定义JavaScript `Person` 类型，带有属性 `firstName` 和 `lastName` ，以及方法 `getFullName` 。

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.getFullName = function() {
        return this.firstName + " " + this.lastName;
    }
}
```

JavaScript方法 `getFullName` 可以通过 `callMember()` 在 `ScriptObjectMirror` 上调用。

```
static void fun4(ScriptObjectMirror person) {
    System.out.println("Full Name is: " + person.callMember("get
FullName"));
}
```

当向Java方法传递新的 `Person` 时，我们会在控制台看到预期的结果：

```
var person1 = new Person("Peter", "Parker");
MyJavaClass.fun4(person1);

// Full Name is: Peter Parker
```

语言扩展

Nashorn定义了多种对ECMAScript标准的语言和API扩展。让我们看一看最新的特性：

类型数组

JavaScript的原生数组是无类型的。Nashron允许你在JavaScript中使用Java的类型数组：

```
var IntArray = Java.type("int[]");

var array = new IntArray(5);
array[0] = 5;
array[1] = 4;
array[2] = 3;
array[3] = 2;
array[4] = 1;

try {
    array[5] = 23;
} catch (e) {
    print(e.message);  // Array index out of range: 5
}

array[0] = "17";
print(array[0]);  // 17

array[0] = "wrong type";
print(array[0]);  // 0

array[0] = "17.3";
print(array[0]);  // 17
```

`int[]` 数组就像真实的Java整数数组那样。但是此外，在我们试图向数组添加非整数时，Nashron在背后执行了一些隐式的转换。字符串会自动转换为整数，这十分便利。

集合和范围遍历

我们可以使用任何Java集合，而避免使用数组瞎折腾。首先需要通过 `Java.type` 定义Java类型，之后创建新的实例。

```
var ArrayList = Java.type('java.util.ArrayList');
var list = new ArrayList();
list.add('a');
list.add('b');
list.add('c');

for each (var el in list) print(el);  // a, b, c
```

为了迭代集合和数组，Nashron引入了 `for each` 语句。它就像Java的范围遍历那样工作。

下面是另一个集合的范围遍历示例，使用 `HashMap` ：

```
var map = new java.util.HashMap();
map.put('foo', 'val1');
map.put('bar', 'val2');

for each (var e in map.keySet()) print(e);  // foo, bar

for each (var e in map.values()) print(e);  // val1, val2
```

Lambda表达式和数据流

每个人都热爱lambda和数据流 -- Nashron也一样！虽然ECMAScript 5.1没有Java8 lmbda表达式的简化箭头语法，我们可以在任何接受lambda表达式的地方使用函数字面值。

```
var list2 = new java.util.ArrayList();
list2.add("ddd2");
list2.add("aaa2");
list2.add("bbb1");
list2.add("aaa1");
list2.add("bbb3");
list2.add("ccc");
list2.add("bbb2");
list2.add("ddd1");

list2
    .stream()
    .filter(function(el) {
        return el.startsWith("aaa");
    })
    .sorted()
    .forEach(function(el) {
        print(el);
    });
    // aaa1, aaa2
```

类的继承

Java类型可以由 `Java.extend` 轻易扩展。就像你在下面的例子中看到的那样，你甚至可以在你的脚本中创建多线程的代码：

```
var Runnable = Java.type('java.lang.Runnable');
var Printer = Java.extend(Runnable, {
    run: function() {
        print('printed from a separate thread');
    }
});

var Thread = Java.type('java.lang.Thread');
new Thread(new Printer()).start();

new Thread(function() {
    print('printed from another thread');
}).start();

// printed from a separate thread
// printed from another thread
```

参数重载

方法和函数可以通过点运算符或方括号运算符来调用：

```
var System = Java.type('java.lang.System');
System.out.println(10);              // 10
System.out["println"](11.0);         // 11.0
System.out["println(double)"](12);   // 12.0
```

当使用重载参数调用方法时，传递可选参数类型 `println(double)` 会指定所调用的具体方法。

Java Beans

你可以简单地使用属性名称来向Java Beans获取或设置值，不需要显式调用读写器：

```
var Date = Java.type('java.util.Date');
var date = new Date();
date.year += 1900;
print(date.year);  // 2014
```

函数字面值

对于简单的单行函数，我们可以去掉花括号：

```
function sqr(x) x * x;
print(sqr(3));    // 9
```

属性绑定

两个不同对象的属性可以绑定到一起：

```
var o1 = {};
var o2 = { foo: 'bar'};

Object.bindProperties(o1, o2);

print(o1.foo);    // bar
o1.foo = 'BAM';
print(o2.foo);    // BAM
```

字符串去空白

我喜欢去掉空白的字符串：

```
print("   hehe".trimLeft());            // hehe
print("hehe    ".trimRight() + "he");   // hehehe
```

位置

以防你忘了自己在哪里：

```
print(__FILE__, __LINE__, __DIR__);
```

导入作用域

有时一次导入多个Java包会很方便。我们可以使用 `JavaImporter` 类，和 `with` 语句一起使用。所有被导入包的类文件都可以在 `with` 语句的局部域中访问到。

```
var imports = new JavaImporter(java.io, java.lang);
with (imports) {
    var file = new File(__FILE__);
    System.out.println(file.getAbsolutePath());
    // /path/to/my/script.js
}
```

数组转换

一些类似 `java.util` 的包可以不使用 `java.type` 或 `JavaImporter` 直接访问：

```
var list = new java.util.ArrayList();
list.add("s1");
list.add("s2");
list.add("s3");
```

下面的代码将Java列表转换为JavaScript原生数组：

```
var jsArray = Java.from(list);
print(jsArray);                                  // s1,s2,s3
print(Object.prototype.toString.call(jsArray));  // [object Arra
y]
```

下面的代码执行相反操作：

```
var javaArray = Java.to([3, 5, 7, 11], "int[]");
```

访问超类

在JavaScript中访问被覆盖的成员通常比较困难，因为Java的 `super` 关键字在ECMAScript中并不存在。幸运的是，Nashron有一套补救措施。

首先我们需要在Java代码中定义超类：

```
class SuperRunner implements Runnable {
    @Override
    public void run() {
        System.out.println("super run");
    }
}
```

下面我在JavaScript中覆盖了 `SuperRunner` 。要注意创建新的 `Runner` 实例时的Nashron语法：覆盖成员的语法取自Java的匿名对象。

```
var SuperRunner = Java.type('com.winterbe.java8.SuperRunner');
var Runner = Java.extend(SuperRunner);

var runner = new Runner() {
    run: function() {
        Java.super(runner).run();
        print('on my run');
    }
}
runner.run();

// super run
// on my run
```

我们通过 `Java.super()` 扩展调用了被覆盖的 `SuperRunner.run()` 方法。

加载脚本

在JavaScript中加载额外的脚本文件非常方便。我们可以使用 `load` 函数加载本地或远程脚本。

我在我的Web前端中大量使用Underscore.js，所以让我们在Nashron中复用它：

```
load('http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.6.0/
underscore-min.js');

var odds = _.filter([1, 2, 3, 4, 5, 6], function (num) {
    return num % 2 == 1;
});

print(odds);  // 1, 3, 5
```

外部脚本会在相同JavaScript上下文中被执行，所以我们可以直接访问underscore的对象。要记住当变量名称互相冲突时，脚本的加载可能会使你的代码崩溃。

这一问题可以通过把脚本文件加载到新的全局上下文来绕过：

```
loadWithNewGlobal('script.js');
```

命令行脚本

如果你对编写命令行（shell）脚本感兴趣，来试一试Nake吧。Nake是一个Java 8 Nashron的简化构建工具。你只需要在项目特定的 `Nakefile` 中定义任务，之后通过在命令行键入 `nake -- myTask` 来执行这些任务。任务编写为JavaScript，并且

在Nashron的脚本模式下运行，所以你可以使用你的终端、JDK8 API和任意Java库的全部功能。

对Java开发者来说，编写命令行脚本是前所未有的简单...

到此为止

我希望这个教程对你有所帮助，并且你能够享受Nashron JavaScript引擎之旅。有关Nashron的更多信息，请见这里、这里和这里。使用Nashron编写shell脚本的教程请见这里。

我最近发布了一篇后续文章，关于如何在Nashron中使用Backbone.js模型。如果你想要进一步学习Java8，请阅读我的Java8教程，和我的Java8数据流教程。

这篇Nashron教程中的可运行的源代码托管在Github上。请随意fork我的仓库，或者在Twitter上向我反馈。

请坚持编程！

Java 8 并发教程：线程和执行器

> 原文：Java 8 Concurrency Tutorial: Threads and Executors
>
> 译者：BlankKelly
>
> 来源：Java8并发教程：Threads和Executors

欢迎阅读我的Java8并发教程的第一部分。这份指南将会以简单易懂的代码示例来教给你如何在Java8中进行并发编程。这是一系列教程中的第一部分。在接下来的15分钟，你将会学会如何通过线程，任务（tasks）和 exector services来并行执行代码。

- 第一部分：线程和执行器
- 第二部分：同步和锁
- 第三部分：原子变量和 ConcurrentMap

并发在Java5中首次被引入并在后续的版本中不断得到增强。在这篇文章中介绍的大部分概念同样适用于以前的Java版本。不过我的代码示例聚焦于Java8，大量使用lambda表达式和其他新特性。如果你对lambda表达式不属性，我推荐你首先阅读我的Java 8 教程。

`Thread` 和 `Runnable`

所有的现代操作系统都通过进程和线程来支持并发。进程是通常彼此独立运行的程序的实例，比如，如果你启动了一个Java程序，操作系统产生一个新的进程，与其他程序一起并行执行。在这些进程的内部，我们使用线程并发执行代码，因此，我们可以最大限度的利用CPU可用的核心（core）。

Java从JDK1.0开始执行线程。在开始一个新的线程之前，你必须指定由这个线程执行的代码，通常称为task。这可以通过实现 `Runnable` ——一个定义了一个无返回值无参数的 `run()` 方法的函数接口，如下面的代码所示：

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

因为 `Runnable` 是一个函数接口，所以我们利用lambda表达式将当前的线程名打印到控制台。首先，在开始一个线程前我们在主线程中直接运行runnable。

控制台输出的结果可能像下面这样：

```
Hello main
Hello Thread-0
Done!
```

或者这样：

```
Hello main
Done!
Hello Thread-0
```

由于我们不能预测这个runnable是在打印'done'前执行还是在之后执行。顺序是不确定的，因此在大的程序中编写并发程序是一个复杂的任务。

我们可以将线程休眠确定的时间。在这篇文章接下来的代码示例中我们可以通过这种方法来模拟长时间运行的任务。

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Thread thread = new Thread(runnable);
thread.start();
```

当你运行上面的代码时，你会注意到在第一条打印语句和第二条打印语句之间存在一分钟的延迟。 `TimeUnit` 在处理单位时间时一个有用的枚举类。你可以通过调用 `Thread.sleep(1000)` 来达到同样的目的。

使用 `Thread` 类是很单调的且容易出错。由于并发API在2004年Java5发布的时候才被引入。这些API位于 `java.util.concurrent` 包下，包含很多处理并发编程的有用的类。自从这些并发API引入以来，在随后的新的Java版本发布过程中得到不断的增强，甚至Java8提供了新的类和方法来处理并发。

接下来，让我们走进并发API中最重要的一部——executor services。

Executor

并发API引入了 `ExecutorService` 作为一个在程序中直接使用Thread的高层次的替换方案。Executos支持运行异步任务，通常管理一个线程池，这样一来我们就不需要手动去创建新的线程。在不断地处理任务的过程中，线程池内部线程将会得到复用，因此，在我们可以使用一个executor service来运行和我们想在我们整个程序中执行的一样多的并发任务。

下面是使用executors的第一个代码示例：

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
String threadName = Thread.currentThread().getName();
System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

`Executors` 类提供了便利的工厂方法来创建不同类型的 executor services。在这个示例中我们使用了一个单线程线程池的 executor。

代码运行的结果类似于上一个示例，但是当运行代码时，你会注意到一个很大的差别：Java进程从没有停止！Executors必须显式的停止-否则它们将持续监听新的任务。

`ExecutorService` 提供了两个方法来达到这个目的—— `shutdwon()` 会等待正在执行的任务执行完而 `shutdownNow()` 会终止所有正在执行的任务并立即关闭execuotr。

这是我喜欢的通常关闭executors的方式：

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
    }
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

executor通过等待指定的时间让当前执行的任务终止来“温柔的”关闭executor。在等待最长5分钟的时间后，execuote最终会通过中断所有的正在执行的任务关闭。

`Callable` 和 `Future`

除了 `Runnable` ，executor还支持另一种类型的任务—— `Callable` 。Callables也是类似于runnables的函数接口，不同之处在于，Callable返回一个值。

下面的lambda表达式定义了一个callable：在休眠一分钟后返回一个整数。

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Callbale也可以像runnbales一样提交给 executor services。但是callables的结果怎么办？因为 `submit()` 不会等待任务完成，executor service不能直接返回callable的结果。不过，executor 可以返回一个 `Future` 类型的结果，它可以用来在稍后某个时间取出实际的结果。

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

在将callable提交给exector之后，我们先通过调用 `isDone()` 来检查这个future是否已经完成执行。我十分确定这会发生什么，因为在返回那个整数之前callable会休眠一分钟、

在调用 `get()` 方法时，当前线程会阻塞等待，直到callable在返回实际的结果123之前执行完成。现在future执行完毕，我们可以在控制台看到如下的结果：

```
future done? false
future done? true
result: 123
```

Future与底层的executor service紧密的结合在一起。记住，如果你关闭executor，所有的未中止的future都会抛出异常。

```
executor.shutdownNow();
future.get();
```

你可能注意到我们这次创建executor的方式与上一个例子稍有不同。我们使用 `newFixedThreadPool(1)` 来创建一个单线程线程池的 execuot service。 这等同于使用 `newSingleThreadExecutor` 不过使用第二种方式我们可以稍后通过简单的传入一个比1大的值来增加线程池的大小。

超时

任何 `future.get()` 调用都会阻塞，然后等待直到callable中止。在最糟糕的情况下，一个callable持续运行——因此使你的程序将没有响应。我们可以简单的传入一个时长来避免这种情况。

```
    ExecutorService executor = Executors.newFixedThreadPool(1);

    Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
});

    future.get(1, TimeUnit.SECONDS);
```

运行上面的代码将会产生一个 `TimeoutException` ：

```
Exception in thread "main" java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask.get(FutureTask.java:205)
```

你可能已经猜到俄为什么会排除这个异常。我们指定的最长等待时间为1分钟，而这个callable在返回结果之前实际需要两分钟。

`invokeAll`

Executors支持通过 `invokeAll()` 一次批量提交多个callable。这个方法结果一个callable的集合，然后返回一个future的列表。

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");

executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

在这个例子中，我们利用Java8中的函数流（stream）来处理 `invokeAll()` 调用返回的所有future。我们首先将每一个future映射到它的返回值，然后将每个值打印到控制台。如果你还不属性stream，可以阅读我的Java8 Stream 教程。

invokeAny

批量提交callable的另一种方式就是 `invokeAny()` ，它的工作方式与 `invokeAll()` 稍有不同。在等待future对象的过程中，这个方法将会阻塞直到第一个callable中止然后返回这一个callable的结果。

为了测试这种行为，我们利用这个帮助方法来模拟不同执行时间的callable。这个方法返回一个callable，这个callable休眠指定 的时间直到返回给定的结果。

```
Callable<String> callable(String result, long sleepSeconds) {
    return () -> {
        TimeUnit.SECONDS.sleep(sleepSeconds);
        return result;
    };
}
```

我们利用这个方法创建一组callable，这些callable拥有不同的执行时间，从1分钟到3分钟。通过 `invokeAny()` 将这些callable提交给一个executor，返回最快的callable的字符串结果-在这个例子中为任务2：

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
callable("task1", 2),
callable("task2", 1),
callable("task3", 3));

String result = executor.invokeAny(callables);
System.out.println(result);

// => task2
```

上面这个例子又使用了另一种方式来创建executor——调用 `newWorkStealingPool()` 。这个工厂方法是Java8引入的，返回一个 `ForkJoinPool` 类型的 executor，它的工作方法与其他常见的execuotr稍有不同。与使用一个固定大小的线程池不同， `ForkJoinPools` 使用一个并行因子数来创建，默认值为主机CPU的可用核心数。

ForkJoinPools 在Java7时引入，将会在这个系列后面的教程中详细讲解。让我们深入了解一下 scheduled executors 来结束本次教程。

ScheduledExecutor

我们已经学习了如何在一个 executor 中提交和运行一次任务。为了持续的多次执行常见的任务，我们可以利用调度线程池。

`ScheduledExecutorService` 支持任务调度，持续执行或者延迟一段时间后执行。

下面的实例，调度一个任务在延迟3分钟后执行：

```
ScheduledExecutorService executor =                 Executors.ne
wScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System
.nanoTime());
ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.
SECONDS);

TimeUnit.MILLISECONDS.sleep(1337);

long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
System.out.printf("Remaining Delay: %sms", remainingDelay);
```

调度一个任务将会产生一个专门的future类型—— `ScheduleFuture` ，它除了提供了Future的所有方法之外，他还提供了 `getDelay()` 方法来获得剩余的延迟。在延迟消逝后，任务将会并发执行。

为了调度任务持续的执行，executors 提供了两个方法 `scheduleAtFixedRate()` 和 `scheduleWithFixedDelay()` 。第一个方法用来以固定频率来执行一个任务，比如，下面这个示例中，每分钟一次：

```
ScheduledExecutorService executor =     Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());

int initialDelay = 0;
int period = 1;
executor.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.SECONDS);
```

另外，这个方法还接收一个初始化延迟，用来指定这个任务首次被执行等待的时长。

请记住： `scheduleAtFixedRate()` 并不考虑任务的实际用时。所以，如果你指定了一个period为1分钟而任务需要执行2分钟，那么线程池为了性能会更快的执行。

在这种情况下，你应该考虑使用 `scheduleWithFixedDelay()` 。这个方法的工作方式与上我们上面描述的类似。不同之处在于等待时间 period 的应用是在一次任务的结束和下一个任务的开始之间。例如：

```
ScheduledExecutorService executor =         Executors.newScheduledThreadPool(1);

Runnable task = () -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        System.out.println("Scheduling: " + System.nanoTime());
    }
    catch (InterruptedException e) {
        System.err.println("task interrupted");
    }
};

executor.scheduleWithFixedDelay(task, 0, 1, TimeUnit.SECONDS);
```

这个例子调度了一个任务，并在一次执行的结束和下一次执行的开始之间设置了一个1分钟的固定延迟。初始化延迟为0，任务执行时间为0。所以我们分别在0s,3s,6s,9s等间隔处结束一次执行。如你所见， `scheduleWithFixedDelay()` 在你不能预测调度任务的执行时长时是很有用的。

这是并发系列教程的第一部分。我推荐你亲手实践一下上面的代码示例。你可以从 Github 上找到这篇文章中所有的代码示例，所以欢迎你fork这个仓库，并收藏它。

我希望你会喜欢这篇文章。如果你有任何的问题都可以在下面评论或者通过 Twitter 向我反馈。

- 第一部分：线程和执行器
- 第二部分：同步和锁
- 第三部分：原子变量和 ConcurrentMap

Java 8 并发教程：同步和锁

> 原文：Java 8 Concurrency Tutorial: Synchronization and Locks
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

欢迎阅读我的Java8并发教程的第二部分。这份指南将会以简单易懂的代码示例来教给你如何在Java8中进行并发编程。这是一系列教程中的第二部分。在接下来的15分钟，你将会学会如何通过同步关键字，锁和信号量来同步访问共享可变变量。

- 第一部分：线程和执行器
- 第二部分：同步和锁
- 第三部分：原子变量和 ConcurrentMap

这篇文章中展示的中心概念也适用于Java的旧版本，然而代码示例适用于Java 8，并严重依赖于lambda表达式和新的并发特性。如果你还不熟悉lambda，我推荐你先阅读我的Java 8 教程。

出于简单的因素，这个教程的代码示例使用了定义在这里的两个辅助函数 `sleep(seconds)` 和 `stop(executor)` 。

同步

在上一章中，我们学到了如何通过执行器服务同时执行代码。当我们编写这种多线程代码时，我们需要特别注意共享可变变量的并发访问。假设我们打算增加某个可被多个线程同时访问的整数。

我们定义了 `count` 字段，带有 `increment()` 方法来使 `count` 加一：

```
int count = 0;

void increment() {
    count = count + 1;
}
```

当多个线程并发调用这个方法时，我们就会遇到大麻烦：

```java
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)
    .forEach(i -> executor.submit(this::increment));

stop(executor);

System.out.println(count);  // 9965
```

我们没有看到 `count` 为10000的结果，上面代码的实际结果在每次执行时都不同。原因是我们在不同的线程上共享可变变量，并且变量访问没有同步机制，这会产生竞争条件。

增加一个数值需要三个步骤：（1）读取当前值，（2）使这个值加一，（3）将新的值写到变量。如果两个线程同时执行，就有可能出现两个线程同时执行步骤1，于是会读到相同的当前值。这会导致无效的写入，所以实际的结果会偏小。上面的例子中，对 `count` 的非同步并发访问丢失了35次增加操作，但是你在自己执行代码时会看到不同的结果。

幸运的是，Java自从很久之前就通过 `synchronized` 关键字支持线程同步。我们可以使用 `synchronized` 来修复上面在增加 `count` 时的竞争条件。

```java
synchronized void incrementSync() {
    count = count + 1;
}
```

在我们并发调用 `incrementSync()` 时，我们得到了 `count` 为10000的预期结果。没有再出现任何竞争条件，并且结果在每次代码执行中都很稳定：

```java
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)
    .forEach(i -> executor.submit(this::incrementSync));

stop(executor);

System.out.println(count);  // 10000
```

`synchronized` 关键字也可用于语句块：

```java
void incrementSync() {
    synchronized (this) {
        count = count + 1;
    }
}
```

Java在内部使用所谓的“监视器”（monitor），也称为监视器锁（monitor lock）或内在锁（ intrinsic lock）来管理同步。监视器绑定在对象上，例如，当使用同步方法时，每个方法都共享相应对象的相同监视器。

所有隐式的监视器都实现了重入（reentrant）特性。重入的意思是锁绑定在当前线程上。线程可以安全地多次获取相同的锁，而不会产生死锁（例如，同步方法调用相同对象的另一个同步方法）。

锁

并发API支持多种显式的锁，它们由 `Lock` 接口规定，用于代替 `synchronized` 的隐式锁。锁对细粒度的控制支持多种方法，因此它们比隐式的监视器具有更大的开销。

锁的多个实现在标准JDK中提供，它们会在下面的章节中展示。

`ReentrantLock`

`ReentrantLock` 类是互斥锁，与通过 `synchronized` 访问的隐式监视器具有相同行为，但是具有扩展功能。就像它的名称一样，这个锁实现了重入特性，就像隐式监视器一样。

让我们看看使用 `ReentrantLock` 之后的上面的例子。

```
ReentrantLock lock = new ReentrantLock();
int count = 0;

void increment() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();
    }
}
```

锁可以通过 `lock()` 来获取，通过 `unlock()` 来释放。把你的代码包装在 `try-finally` 代码块中来确保异常情况下的解锁非常重要。这个方法是线程安全的，就像同步副本那样。如果另一个线程已经拿到锁了，再次调用 `lock()` 会阻塞当前线程，直到锁被释放。在任意给定的时间内，只有一个线程可以拿到锁。

锁对细粒度的控制支持多种方法，就像下面的例子那样：

```
executor.submit(() -> {
    lock.lock();
    try {
        sleep(1);
    } finally {
        lock.unlock();
    }
});

executor.submit(() -> {
    System.out.println("Locked: " + lock.isLocked());
    System.out.println("Held by me: " + lock.isHeldByCurrentThre
ad());
    boolean locked = lock.tryLock();
    System.out.println("Lock acquired: " + locked);
});

stop(executor);
```

在第一个任务拿到锁的一秒之后，第二个任务获得了锁的当前状态的不同信息。

```
Locked: true
Held by me: false
Lock acquired: false
```

`tryLock()` 方法是 `lock()` 方法的替代，它尝试拿锁而不阻塞当前线程。在访问任何共享可变变量之前，必须使用布尔值结果来检查锁是否已经被获取。

`ReadWriteLock`

`ReadWriteLock` 接口规定了锁的另一种类型，包含用于读写访问的一对锁。读写锁的理念是，只要没有任何线程写入变量，并发读取可变变量通常是安全的。所以读锁可以同时被多个线程持有，只要没有线程持有写锁。这样可以提升性能和吞吐量，因为读取比写入更加频繁。

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
ReadWriteLock lock = new ReentrantReadWriteLock();

executor.submit(() -> {
    lock.writeLock().lock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.writeLock().unlock();
    }
});
```

上面的例子在暂停一秒之后，首先获取写锁来向映射添加新的值。在这个任务完成之前，两个其它的任务被启动，尝试读取映射中的元素，并暂停一秒：

```
Runnable readTask = () -> {
    lock.readLock().lock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.readLock().unlock();
    }
};

executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

当你执行这一代码示例时，你会注意到两个读任务需要等待写任务完成。在释放了写锁之后，两个读任务会同时执行，并同时打印结果。它们不需要相互等待完成，因为读锁可以安全同步获取，只要没有其它线程获取了写锁。

`StampedLock`

Java 8 自带了一种新的锁，叫做 `StampedLock` ，它同样支持读写锁，就像上面的例子那样。与 `ReadWriteLock` 不同的是， `StampedLock` 的锁方法会返回表示为 `long` 的标记。你可以使用这些标记来释放锁，或者检查锁是否有效。此外， `StampedLock` 支持另一种叫做乐观锁（optimistic locking）的模式。

让我们使用 `StampedLock` 代替 `ReadWriteLock` 重写上面的例子：

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.unlockWrite(stamp);
    }
});

Runnable readTask = () -> {
    long stamp = lock.readLock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.unlockRead(stamp);
    }
};

executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

通过 `readLock()` 或 `writeLock()` 来获取读锁或写锁会返回一个标记，它可以在稍后用于在 `finally` 块中解锁。要记住 `StampedLock` 并没有实现重入特性。每次调用加锁都会返回一个新的标记，并且在没有可用的锁时阻塞，即使相同线程已经拿锁了。所以你需要额外注意不要出现死锁。

就像前面的 `ReadWriteLock` 例子那样，两个读任务都需要等待写锁释放。之后两个读任务同时向控制台打印信息，因为多个读操作不会相互阻塞，只要没有线程拿到写锁。

下面的例子展示了乐观锁：

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.tryOptimisticRead();
    try {
        System.out.println("Optimistic Lock Valid: " + lock.vali
date(stamp));
        sleep(1);
        System.out.println("Optimistic Lock Valid: " + lock.vali
date(stamp));
        sleep(2);
        System.out.println("Optimistic Lock Valid: " + lock.vali
date(stamp));
    } finally {
        lock.unlock(stamp);
    }
});

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        System.out.println("Write Lock acquired");
        sleep(2);
    } finally {
        lock.unlock(stamp);
        System.out.println("Write done");
    }
});

stop(executor);
```

乐观的读锁通过调用 `tryOptimisticRead()` 获取，它总是返回一个标记而不阻塞当前线程，无论锁是否真正可用。如果已经有写锁被拿到，返回的标记等于0。你需要总是通过 `lock.validate(stamp)` 检查标记是否有效。

执行上面的代码会产生以下输出：

```
Optimistic Lock Valid: true
Write Lock acquired
Optimistic Lock Valid: false
Write done
Optimistic Lock Valid: false
```

乐观锁在刚刚拿到锁之后是有效的。和普通的读锁不同的是，乐观锁不阻止其他线程同时获取写锁。在第一个线程暂停一秒之后，第二个线程拿到写锁而无需等待乐观的读锁被释放。此时，乐观的读锁就不再有效了。甚至当写锁释放时，乐观的读锁还处于无效状态。

所以在使用乐观锁时，你需要每次在访问任何共享可变变量之后都要检查锁，来确保读锁仍然有效。

有时，将读锁转换为写锁而不用再次解锁和加锁十分实用。 `StampedLock` 为这种目的提供了 `tryConvertToWriteLock()` 方法，就像下面那样：

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.readLock();
    try {
        if (count == 0) {
            stamp = lock.tryConvertToWriteLock(stamp);
            if (stamp == 0L) {
                System.out.println("Could not convert to write l
ock");
                stamp = lock.writeLock();
            }
            count = 23;
        }
        System.out.println(count);
    } finally {
        lock.unlock(stamp);
    }
});

stop(executor);
```

第一个任务获取读锁，并向控制台打印 `count` 字段的当前值。但是如果当前值是零，我们希望将其赋值为 `23` 。我们首先需要将读锁转换为写锁，来避免打破其它线程潜在的并发访问。 `tryConvertToWriteLock()` 的调用不会阻塞，但是可能会返回为零的标记，表示当前没有可用的写锁。这种情况下，我们调用 `writeLock()` 来阻塞当前线程，直到有可用的写锁。

信号量

除了锁之外，并发API也支持计数的信号量。不过锁通常用于变量或资源的互斥访问，信号量可以维护整体的准入许可。这在一些不同场景下，例如你需要限制你程序某个部分的并发访问总数时非常实用。

下面是一个例子，演示了如何限制对通过 `sleep(5)` 模拟的长时间运行任务的访问：

```
ExecutorService executor = Executors.newFixedThreadPool(10);

Semaphore semaphore = new Semaphore(5);

Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            sleep(5);
        } else {
            System.out.println("Could not acquire semaphore");
        }
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {
            semaphore.release();
        }
    }
}

IntStream.range(0, 10)
    .forEach(i -> executor.submit(longRunningTask));

stop(executor);
```

执行器可能同时运行10个任务，但是我们使用了大小为5的信号量，所以将并发访问限制为5。使用 `try-finally` 代码块在异常情况中合理释放信号量十分重要。

执行上述代码产生如下结果：

```
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
```

信号量限制对通过 `sleep(5)` 模拟的长时间运行任务的访问，最大5个线程。每个随后的 `tryAcquire()` 调用在经过最大为一秒的等待超时之后，会向控制台打印不能获取信号量的结果。

这就是我的系列并发教程的第二部分。以后会放出更多的部分，所以敬请等待吧。像以前一样，你可以在Github上找到这篇文档的所有示例代码，所以请随意fork这个仓库，并自己尝试它。

我希望你能喜欢这篇文章。如果你还有任何问题，在下面的评论中向我反馈。你也可以在Twitter上关注我来获取更多开发相关的信息。

- 第一部分：线程和执行器
- 第二部分：同步和锁
- 第三部分：原子变量和 ConcurrentMap

Java 8 并发教程：原子变量和 ConcurrentMap

> 原文：Java 8 Concurrency Tutorial: Synchronization and Locks
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

欢迎阅读我的Java8多线程编程系列教程的第三部分。这个教程包含并发API的两个重要部分：原子变量和 `ConcurrentMap` 。由于最近发布的Java8中的lambda表达式和函数式编程，二者都有了极大的改进。所有这些新特性会以一些简单易懂的代码示例来描述。希望你能喜欢。

- 第一部分：线程和执行器
- 第二部分：同步和锁
- 第三部分：原子变量和 ConcurrentMap

出于简单的因素，这个教程的代码示例使用了定义在这里的两个辅助函数 `sleep(seconds)` 和 `stop(executor)` 。

`AtomicInteger`

`java.concurrent.atomic` 包包含了许多实用的类，用于执行原子操作。如果你能够在多线程中同时且安全地执行某个操作，而不需要 `synchronized` 关键字或上一章中的锁，那么这个操作就是原子的。

本质上，原子操作严重依赖于比较与交换（CAS），它是由多数现代CPU直接支持的原子指令。这些指令通常比同步块要快。所以在只需要并发修改单个可变变量的情况下，我建议你优先使用原子类，而不是上一章展示的锁。

> 译者注：对于其它语言，一些语言的原子操作用锁实现，而不是原子指令。

现在让我们选取一个原子类，例如 `AtomicInteger` ：

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> executor.submit(atomicInt::incrementAndGet));

stop(executor);

System.out.println(atomicInt.get());    // => 1000
```

通过使用 `AtomicInteger` 代替 `Integer` ，我们就能线程安全地并发增加数值，而不需要同步访问变量。 `incrementAndGet()` 方法是原子操作，所以我们可以在多个线程中安全调用它。

`AtomicInteger` 支持多种原子操作。 `updateAndGet()` 接受lambda表达式，以便在整数上执行任意操作：

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.updateAndGet(n -> n + 2);
        executor.submit(task);
    });

stop(executor);

System.out.println(atomicInt.get());    // => 2000
```

`accumulateAndGet()` 方法接受另一种类型 `IntBinaryOperator` 的lambda表达式。我们在下个例子中，使用这个方法并发计算0~1000所有值的和：

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.accumulateAndGet(i, (n, m) -> n + m);
        executor.submit(task);
    });

stop(executor);

System.out.println(atomicInt.get());    // => 499500
```

其它实用的原子类有 `AtomicBoolean` 、 `AtomicLong` 和 `AtomicReference` 。

`LongAdder`

`LongAdder` 是 `AtomicLong` 的替代，用于向某个数值连续添加值。

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> executor.submit(adder::increment));

stop(executor);

System.out.println(adder.sumThenReset());   // => 1000
```

`LongAdder` 提供了 `add()` 和 `increment()` 方法，就像原子数值类一样，同样是线程安全的。但是这个类在内部维护一系列变量来减少线程之间的争用，而不是求和计算单一结果。实际的结果可以通过调用 `sum()` 或 `sumThenReset()` 来获取。

当多线程的更新比读取更频繁时，这个类通常比原子数值类性能更好。这种情况在抓取统计数据时经常出现，例如，你希望统计Web服务器上请求的数量。`LongAdder` 缺点是较高的内存开销，因为它在内存中储存了一系列变量。

`LongAccumulator`

`LongAccumulator` 是 `LongAdder` 的更通用的版本。`LongAccumulator` 以类型为 `LongBinaryOperator` lambda表达式构建，而不是仅仅执行加法操作，像这段代码展示的那样：

```
LongBinaryOperator op = (x, y) -> 2 * x + y;
LongAccumulator accumulator = new LongAccumulator(op, 1L);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10)
    .forEach(i -> executor.submit(() -> accumulator.accumulate(i
)));

stop(executor);

System.out.println(accumulator.getThenReset());     // => 2539
```

我们使用函数 `2 * x + y` 创建了 `LongAccumulator`，初始值为1。每次调用 `accumulate(i)` 的时候，当前结果和值 `i` 都会作为参数传入lambda表达式。

`LongAccumulator` 就像 `LongAdder` 那样，在内部维护一系列变量来减少线程之间的争用。

`ConcurrentMap`

`ConcurrentMap` 接口继承自 `Map` 接口，并定义了最实用的并发集合类型之一。Java8通过将新的方法添加到这个接口，引入了函数式编程。

在下面的代码中，我们使用这个映射示例来展示那些新的方法：

```
ConcurrentMap<String, String> map = new ConcurrentHashMap<>();
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

`forEach()` 方法接受类型为 `BiConsumer` 的lambda表达式，以映射的键和值作为参数传递。它可以作为 `for-each` 循环的替代，来遍历并发映射中的元素。迭代在当前线程上串行执行。

```
map.forEach((key, value) -> System.out.printf("%s = %s\n", key,
value));
```

新方法 `putIfAbsent()` 只在提供的键不存在时，将新的值添加到映射中。至少在 `ConcurrentHashMap` 的实现中，这一方法像 `put()` 一样是线程安全的，所以你在不同线程中并发访问映射时，不需要任何同步机制。

```
String value = map.putIfAbsent("c3", "p1");
System.out.println(value);    // p0
```

`getOrDefault()` 方法返回指定键的值。在传入的键不存在时，会返回默认值：

```
String value = map.getOrDefault("hi", "there");
System.out.println(value);    // there
```

`replaceAll()` 接受类型为 `BiFunction` 的lambda表达式。`BiFunction` 接受两个参数并返回一个值。函数在这里以每个元素的键和值调用，并返回要映射到当前键的新值。

```
map.replaceAll((key, value) -> "r2".equals(key) ? "d3" : value);
System.out.println(map.get("r2"));    // d3
```

`compute()` 允许我们转换单个元素，而不是替换映射中的所有值。这个方法接受需要处理的键，和用于指定值的转换的 `BiFunction` 。

```
map.compute("foo", (key, value) -> value + value);
System.out.println(map.get("foo"));   // barbar
```

除了 `compute()` 之外还有两个变体： `computeIfAbsent()` 和 `computeIfPresent()` 。这些方法的函数式参数只在键不存在或存在时被调用。

最后， `merge()` 方法可以用于以映射中的现有值来统一新的值。这个方法接受键、需要并入现有元素的新值，以及指定两个值的合并行为的 `BiFunction` 。

```
map.merge("foo", "boo", (oldVal, newVal) -> newVal + " was " + o
ldVal);
System.out.println(map.get("foo"));   // boo was foo
```

ConcurrentHashMap

所有这些方法都是 `ConcurrentMap` 接口的一部分，因此可在所有该接口的实现上调用。此外，最重要的实现 `ConcurrentHashMap` 使用了一些新的方法来改进，便于在映射上执行并行操作。

就像并行流那样，这些方法使用特定的 `ForkJoinPool` ，由Java8中的 `ForkJoinPool.commonPool()` 提供。该池使用了取决于可用核心数量的预置并行机制。我的电脑有四个核心可用，这会使并行性的结果为3：

```
System.out.println(ForkJoinPool.getCommonPoolParallelism());  //
 3
```

这个值可以通过设置下列JVM参数来增减：

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

我们使用相同的映射示例来展示，但是这次我们使用具体的 `ConcurrentHashMap` 实现而不是 `ConcurrentMap` 接口，所以我们可以访问这个类的所有公共方法：

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>(
);
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

Java8引入了三种类型的并行操作： `forEach` 、 `search` 和 `reduce` 。这些操作中每个都以四种形式提供，接受以键、值、元素或键值对为参数的函数。

所有这些方法的第一个参数是通用的 `parallelismThreshold` 。这一阈值表示操作并行执行时的最小集合大小。例如，如果你传入阈值500，而映射的实际大小是499，那么操作就会在单线程上串行执行。在下一个例子中，我们使用阈值1，始终

强制并行执行来展示。

forEach

`forEach()` 方法可以并行迭代映射中的键值对。 `BiConsumer` 以当前迭代元素的键和值调用。为了将并行执行可视化，我们向控制台打印了当前线程的名称。要注意在我这里底层的 `ForkJoinPool` 最多使用三个线程。

```
map.forEach(1, (key, value) ->
    System.out.printf("key: %s; value: %s; thread: %s\n",
        key, value, Thread.currentThread().getName()));

// key: r2; value: d2; thread: main
// key: foo; value: bar; thread: ForkJoinPool.commonPool-worker-1

// key: han; value: solo; thread: ForkJoinPool.commonPool-worker
-2
// key: c3; value: p0; thread: main
```

search

`search()` 方法接受 `BiFunction` 并为当前的键值对返回一个非空的搜索结果，或者在当前迭代不匹配任何搜索条件时返回 `null` 。只要返回了非空的结果，就不会往下搜索了。要记住 `ConcurrentHashMap` 是无序的。搜索函数应该不依赖于映射实际的处理顺序。如果映射的多个元素都满足指定搜索函数，结果是非确定的。

```
String result = map.search(1, (key, value) -> {
    System.out.println(Thread.currentThread().getName());
    if ("foo".equals(key)) {
        return value;
    }
    return null;
});
System.out.println("Result: " + result);

// ForkJoinPool.commonPool-worker-2
// main
// ForkJoinPool.commonPool-worker-3
// Result: bar
```

下面是另一个例子，仅仅搜索映射中的值：

```
String result = map.searchValues(1, value -> {
    System.out.println(Thread.currentThread().getName());
    if (value.length() > 3) {
        return value;
    }
    return null;
});

System.out.println("Result: " + result);

// ForkJoinPool.commonPool-worker-2
// main
// main
// ForkJoinPool.commonPool-worker-1
// Result: solo
```

reduce

`reduce()` 方法已经在Java 8 的数据流之中用过了，它接受两个 `BiFunction` 类型的lambda表达式。第一个函数将每个键值对转换为任意类型的单一值。第二个函数将所有这些转换后的值组合为单一结果，并忽略所有可能的 `null` 值。

```
String result = map.reduce(1,
    (key, value) -> {
        System.out.println("Transform: " + Thread.currentThread(
).getName());
        return key + "=" + value;
    },
    (s1, s2) -> {
        System.out.println("Reduce: " + Thread.currentThread().g
etName());
        return s1 + ", " + s2;
    });

System.out.println("Result: " + result);

// Transform: ForkJoinPool.commonPool-worker-2
// Transform: main
// Transform: ForkJoinPool.commonPool-worker-3
// Reduce: ForkJoinPool.commonPool-worker-3
// Transform: main
// Reduce: main
// Reduce: main
// Result: r2=d2, c3=p0, han=solo, foo=bar
```

我希望你能喜欢我的Java8并发系列教程的第三部分。这个教程的代码示例托管在Github上，还有许多其它的Java8代码片段。欢迎fork我的仓库并自己尝试。

如果你想要支持我的工作，请向你的朋友分享这篇教程。你也可以在Twiiter上关注我，因为我会不断推送一些Java或编程相关的东西。

- 第一部分：线程和执行器
- 第二部分：同步和锁
- 第三部分：原子变量和 ConcurrentMap

Java 8 API 示例：字符串、数值、算术和文件

> 原文：Java 8 API by Example: Strings, Numbers, Math and Files
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

大量的教程和文章都涉及到Java8中最重要的改变，例如lambda表达式和函数式数据流。但是此外许多现存的类在JDK 8 API中也有所改进，带有一些实用的特性和方法。

这篇教程涉及到Java 8 API中的那些小修改 -- 每个都使用简单易懂的代码示例来描述。让我们好好看一看字符串、数值、算术和文件。

处理字符串

两个新的方法可在字符串类上使用： `join` 和 `chars` 。第一个方法使用指定的分隔符，将任何数量的字符串连接为一个字符串。

```
String.join(":", "foobar", "foo", "bar");
// => foobar:foo:bar
```

第二个方法 `chars` 从字符串所有字符创建数据流，所以你可以在这些字符上使用流式操作。

```
"foobar:foo:bar"
    .chars()
    .distinct()
    .mapToObj(c -> String.valueOf((char)c))
    .sorted()
    .collect(Collectors.joining());
// => :abfor
```

不仅仅是字符串，正则表达式模式串也能受益于数据流。我们可以分割任何模式串，并创建数据流来处理它们，而不是将字符串分割为单个字符的数据流，像下面这样：

```
Pattern.compile(":")
    .splitAsStream("foobar:foo:bar")
    .filter(s -> s.contains("bar"))
    .sorted()
    .collect(Collectors.joining(":"));
// => bar:foobar
```

此外，正则模式串可以转换为谓词。这些谓词可以像下面那样用于过滤字符串流：

```
Pattern pattern = Pattern.compile(".*@gmail\\.com");
Stream.of("bob@gmail.com", "alice@hotmail.com")
    .filter(pattern.asPredicate())
    .count();
// => 1
```

上面的模式串接受任何以 `@gmail.com` 结尾的字符串，并且之后用作Java8的 `Predicate` 来过滤电子邮件地址流。

处理数值

Java8添加了对无符号数的额外支持。Java中的数值总是有符号的，例如，让我们来观察 `Integer` ：

`int` 可表示最多 `2 ** 32` 个数。Java中的数值默认为有符号的，所以最后一个二进制数字表示符号（0为正数，1为负数）。所以从十进制的0开始，最大的有符号正整数为 `2 ** 31 - 1` 。

你可以通过 `Integer.MAX_VALUE` 来访问它：

```
System.out.println(Integer.MAX_VALUE);      // 2147483647
System.out.println(Integer.MAX_VALUE + 1);  // -2147483648
```

Java8添加了解析无符号整数的支持，让我们看看它如何工作：

```
long maxUnsignedInt = (1l << 32) - 1;
String string = String.valueOf(maxUnsignedInt);
int unsignedInt = Integer.parseUnsignedInt(string, 10);
String string2 = Integer.toUnsignedString(unsignedInt, 10);
```

就像你看到的那样，现在可以将最大的无符号数 `2 ** 32 - 1` 解析为整数。而且你也可以将这个数值转换回无符号数的字符串表示。

这在之前不可能使用 `parseInt` 完成，就像这个例子展示的那样：

```
try {
    Integer.parseInt(string, 10);
}
catch (NumberFormatException e) {
    System.err.println("could not parse signed int of " + maxUns
ignedInt);
}
```

这个数值不可解析为有符号整数，因为它超出了最大范围 `2 ** 31 - 1` 。

算术运算

`Math` 工具类新增了一些方法来处理数值溢出。这是什么意思呢？我们已经看到了所有数值类型都有最大值。所以当算术运算的结果不能被它的大小装下时，会发生什么呢？

```
System.out.println(Integer.MAX_VALUE);      // 2147483647
System.out.println(Integer.MAX_VALUE + 1);  // -2147483648
```

就像你看到的那样，发生了整数溢出，这通常是我们不愿意看到的。

Java8添加了严格数学运算的支持来解决这个问题。 `Math` 扩展了一些方法，它们全部以 `exact` 结尾，例如 `addExact` 。当运算结果不能被数值类型装下时，这些方法通过抛出 `ArithmeticException` 异常来合理地处理溢出。

```
try {
    Math.addExact(Integer.MAX_VALUE, 1);
}
catch (ArithmeticException e) {
    System.err.println(e.getMessage());
    // => integer overflow
}
```

当尝试通过 `toIntExact` 将长整数转换为整数时，可能会抛出同样的异常：

```
try {
    Math.toIntExact(Long.MAX_VALUE);
}
catch (ArithmeticException e) {
    System.err.println(e.getMessage());
    // => integer overflow
}
```

处理文件

`Files` 工具类首次在Java7中引入，作为NIO的一部分。JDK8 API添加了一些额外的方法，它们可以将文件用于函数式数据流。让我们深入探索一些代码示例。

列出文件

`Files.list` 方法将指定目录的所有路径转换为数据流，便于我们在文件系统的内容上使用类似 `filter` 和 `sorted` 的流操作。

```
try (Stream<Path> stream = Files.list(Paths.get(""))) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> !path.startsWith("."))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("List: " + joined);
}
```

上面的例子列出了当前工作目录的所有文件，之后将每个路径都映射为它的字符串表示。之后结果被过滤、排序，最后连接为一个字符串。如果你还不熟悉函数式数据流，你应该阅读我的Java8数据流教程。

你可能已经注意到，数据流的创建包装在 `try-with` 语句中。数据流实现了 `AutoCloseable` ，并且这里我们需要显式关闭数据流，因为它基于IO操作。

> 返回的数据流是 `DirectoryStream` 的封装。如果需要及时处理文件资源，就应该使用 `try-with` 结构来确保在流式操作完成后，数据流的 `close` 方法被调用。

查找文件

下面的例子演示了如何查找在目录及其子目录下的文件：

```
Path start = Paths.get("");
int maxDepth = 5;
try (Stream<Path> stream = Files.find(start, maxDepth, (path, at
tr) ->
        String.valueOf(path).endsWith(".js"))) {
    String joined = stream
        .sorted()
        .map(String::valueOf)
        .collect(Collectors.joining("; "));
    System.out.println("Found: " + joined);
}
```

`find` 方法接受三个参数：目录路径 `start` 是起始点， `maxDepth` 定义了最大搜索深度。第三个参数是一个匹配谓词，定义了搜索的逻辑。上面的例子中，我们搜索了所有JavaScirpt文件（以 `.js` 结尾的文件名）。

我们可以使用 `Files.walk` 方法来完成相同的行为。这个方法会遍历每个文件，而不需要传递搜索谓词。

```
Path start = Paths.get("");
int maxDepth = 5;
try (Stream<Path> stream = Files.walk(start, maxDepth)) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> path.endsWith(".js"))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("walk(): " + joined);
}
```

这个例子中，我们使用了流式操作 `filter` 来完成和上个例子相同的行为。

读写文件

将文本文件读到内存，以及向文本文件写入字符串在Java 8 中是简单的任务。不需要再去摆弄读写器了。 `Files.readAllLines` 从指定的文件把所有行读进字符串列表中。你可以简单地修改这个列表，并且将它通过 `Files.write` 写到另一个文件中：

```
List<String> lines = Files.readAllLines(Paths.get("res/nashorn1.
js"));
lines.add("print('foobar');");
Files.write(Paths.get("res/nashorn1-modified.js"), lines);
```

要注意这些方法对内存并不十分高效，因为整个文件都会读进内存。文件越大，所用的堆区也就越大。

你可以使用 `Files.lines` 方法来作为内存高效的替代。这个方法读取每一行，并使用函数式数据流来对其流式处理，而不是一次性把所有行都读进内存。

```
try (Stream<String> stream = Files.lines(Paths.get("res/nashorn1
.js"))) {
    stream
        .filter(line -> line.contains("print"))
        .map(String::trim)
        .forEach(System.out::println);
}
```

如果你需要更多的精细控制，你需要构造一个新的 `BufferedReader` 来代替：

```
Path path = Paths.get("res/nashorn1.js");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    System.out.println(reader.readLine());
}
```

或者，你需要写入文件时，简单地构造一个 `BufferedWriter` 来代替：

```
Path path = Paths.get("res/output.js");
try (BufferedWriter writer = Files.newBufferedWriter(path)) {
    writer.write("print('Hello World');");
}
```

`BufferedReader` 也可以访问函数式数据流。 `lines` 方法在它所有行上面构建数据流：

```
Path path = Paths.get("res/nashorn1.js");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    long countPrints = reader
        .lines()
        .filter(line -> line.contains("print"))
        .count();
    System.out.println(countPrints);
}
```

目前为止你可以看到Java8提供了三个简单的方法来读取文本文件的每一行，使文件处理更加便捷。

不幸的是你需要显式使用 `try-with` 语句来关闭文件流，这会使示例代码有些凌乱。我期待函数式数据流可以在调用类似 `count` 和 `collect` 时可以自动关闭，因为你不能在相同数据流上调用终止操作两次。

我希望你能喜欢这篇文章。所有示例代码都托管在Github上，还有来源于我博客其它Java8文章的大量的代码片段。如果这篇文章对你有所帮助，请收藏我的仓库，并且在Twitter上关注我。

请坚持编程！

在 Java 8 中避免 Null 检查

> 原文：Avoid Null Checks in Java 8
>
> 译者：ostatsu
>
> 来源：在 Java 8 中避免 Null 检查

如何预防 Java 中著名的 NullPointerException 异常？这是每个 Java 初学者迟早会问到的关键问题之一。而且中级和高级程序员也在时时刻刻规避这个错误。其是迄今为止 Java 以及很多其他编程语言中最流行的一种错误。

Null 引用的发明者 Tony Hoare 在 2009 年道歉，并称这种错误为他的十亿美元错误。

> 我将其称之为自己的十亿美元错误。它的发明是在1965 年，那时我用一个面向对象语言（ALGOL W）设计了第一个全面的引用类型系统。我的目的是确保所有引用的使用都是绝对安全的，编译器会自动进行检查。但是我未能抵御住诱惑，加入了 Null 引用，仅仅是因为实现起来非常容易。它导致了数不清的错误、漏洞和系统崩溃，可能在之后 40 年中造成了十亿美元的损失。

无论如何，我们必须要面对它。所以，我们到底能做些什么来防止 NullPointerException 异常呢？那么，答案显然是对其添加 null 检查。由于 null 检查还是挺麻烦和痛苦的，很多语言为了处理 null 检查添加了特殊的语法，即空合并运算符 —— 其在像 Groovy 或 Kotlin 这样的语言中也被称为 Elvis 运算符。

不幸的是 Java 没有提供这样的语法糖。但幸运的是这在 Java 8 中得到了改善。这篇文章介绍了如何利用像 lambda 表达式这样的 Java 8 新特性来防止编写不必要的 null 检查的几个技巧。

在 Java 8 中提高 Null 的安全性

我已经在另一篇文章中说明了我们可以如何利用 Java 8 的 Optional 类型来预防 null 检查。下面是那篇文章中的示例代码。

假设我们有一个像这样的类层次结构：

```
class Outer {
    Nested nested;
    Nested getNested() {
        return nested;
    }
}
class Nested {
    Inner inner;
    Inner getInner() {
        return inner;
    }
}
class Inner {
    String foo;
    String getFoo() {
        return foo;
    }
}
```

解决这种结构的深层嵌套路径是有点麻烦的。我们必须编写一堆 null 检查来确保不会导致一个 NullPointerException：

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner
!= null) {
    System.out.println(outer.nested.inner.foo);
}
```

我们可以通过利用 Java 8 的 Optional 类型来摆脱所有这些 null 检查。map 方法接收一个 Function 类型的 lambda 表达式，并自动将每个 function 的结果包装成一个 Optional 对象。这使我们能够在一行中进行多个 map 操作。Null 检查是在底层自动处理的。

```
Optional.of(new Outer())
    .map(Outer::getNested)
    .map(Nested::getInner)
    .map(Inner::getFoo)
    .ifPresent(System.out::println);
```

还有一种实现相同作用的方式就是通过利用一个 supplier 函数来解决嵌套路径的问题：

```
Outer obj = new Outer();
resolve(() -> obj.getNested().getInner().getFoo());
    .ifPresent(System.out::println);
```

调用 obj.getNested().getInner().getFoo()) 可能会抛出一个 NullPointerException 异常。在这种情况下，该异常将会被捕获，而该方法会返回 Optional.empty()。

```
public static <T> Optional<T> resolve(Supplier<T> resolver) {
    try {
        T result = resolver.get();
        return Optional.ofNullable(result);
    }
    catch (NullPointerException e) {
        return Optional.empty();
    }
}
```

请记住，这两个解决方案可能没有传统 null 检查那么高的性能。不过在大多数情况下不会有太大问题。

像往常一样，上面的示例代码都托管在 GitHub。

祝编程愉快！

使用 Intellij IDEA 解决 Java 8 的数据流问题

> 原文：Fixing Java 8 Stream Gotchas with IntelliJ IDEA
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

Java8在2014年三月发布，距离现在（2015年三月五号）快有一年了。我们打算将Pondus的所有生产服务器升级到这一新版本。从那时起，我们将大部分代码库迁移到lambda表达式、数据流和新的日期API上。我们也会使用Nashorn来把我们的应用中运行时发生改变的部分变成动态脚本。

除了lambda，最实用的特性是新的数据流API。集合操作在任何我见过的代码库中都随处可见。而且对于那些集合操作，数据流是提升代码可读性的好方法。

但是一件关于数据流的事情十分令我困扰：数据流只提供了几个终止操作，例如 `reduce` 和 `findFirst` 属于直接操作，其它的只能通过 `collect` 来访问。工具类 `Collctors` 提供了一些便利的收集器，例如 `toList` 、 `toSet` 、 `joining` 和 `groupingBy` 。

例如，下面的代码对一个字符串集合进行过滤，并创建新的列表：

```
stringCollection
    .stream()
    .filter(e -> e.startsWith("a"))
    .collect(Collectors.toList());
```

在迁移了300k行代码到数据流之后，我可以说， `toList` 、 `toSet` 、和 `groupingBy` 是你的项目中最常用的终止操作。所以我不能理解为什么不把这些方法直接集成到 `Stream` 接口上面，这样你就可以直接编写：

```
stringCollection
    .stream()
    .filter(e -> e.startsWith("a"))
    .toList();
```

这在开始看起来是个小缺陷，但是如果你需要一遍又一遍地编写这些代码，它会非常烦人。

有 `toArray()` 方法但是没有 `toList()` ，所以我真心希望一些便利的收集器可以在Java9中这样添加到 `Stream` 接口中。是吧，Brian？□_□

> 注：Stream.js是浏览器上的Java 8 数据流API的JavaScript接口，并解决了上述问题。所有重要的终止操作都可以直接在流上访问，十分方便。详情请见API文档。

无论如何，IntelliJ IDEA声称它是最智能的Java IDE。所以让我们看看如何使用IDEA来解决这一问题。

使用 IntelliJ IDEA 来帮忙

IntelliJ IDEA自带了一个便利的特性，叫做实时模板（Live Template）。如果你还不知道它是什么：实时模板是一些常用代码段的快捷方式。例如，你键入 `sout` 并按下TAB键，IDEA就会插入代码段 `System.out.println()` 。更多信息请见这里。

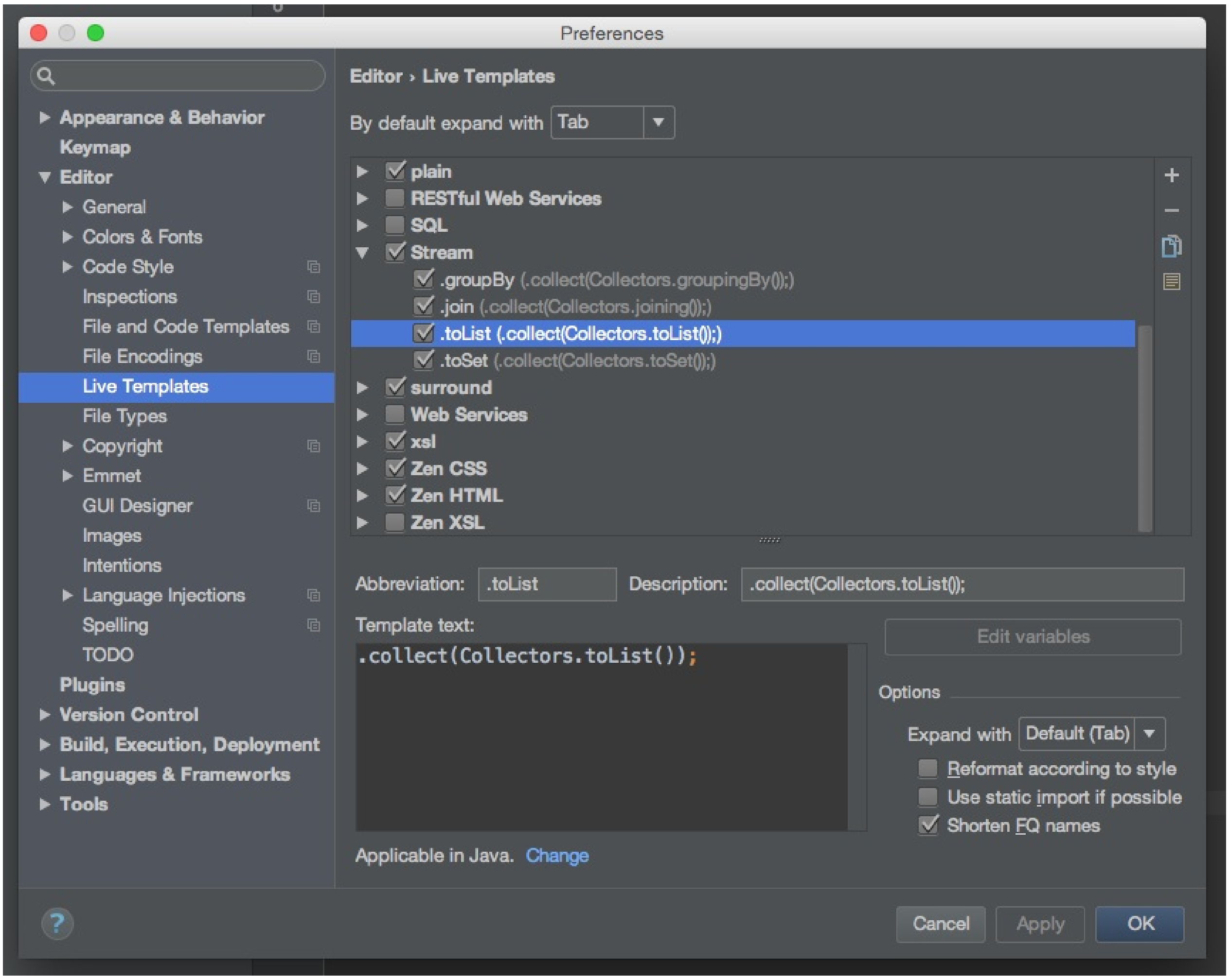
如何用实时模板来解决上述问题？实际上我们只需要为所有普遍使用的默认数据流收集器创建我们自己的实时模板。例如，我们可以创建 `.toList` 缩写的实时模板，来自动插入适当的收集器 `.collect(Collectors.toList())` 。

下面是它在实际工作中的样子：

```
stringCollection
        .stream()
        .filter(s -> s.startsWith("a"))
```

构建你自己的实时模板

让我们看看如何自己构建它。首先访问设置（Settings）并在左侧的菜单中选择实时模板。你也可以使用对话框左上角的便利的输入过滤。



下面我们可以通过右侧的 `+` 图标创建一个新的组，叫做 `Stream` 。接下来我们向组中添加所有数据流相关的实时模板。我经常使用默认的收集器 `toList` 、 `toSet` 、 `groupingBy` 和 `join` ，所以我为每个这些方法都创建了新的实时模板。

这一步非常重要。在添加新的实时模板之后，你需要在对话框底部指定合适的上下文。你需要选择 `Java → Other` ，然后定义缩写、描述和实际的模板代码。

```
// Abbreviation: .toList
.collect(Collectors.toList())

// Abbreviation: .toSet
.collect(Collectors.toSet())

// Abbreviation: .join
.collect(Collectors.joining("$END$"))

// Abbreviation: .groupBy
.collect(Collectors.groupingBy(e -> $END$))
```

特殊的变量 `$END$` 指定在使用模板之后的光标位置，所以你可以直接在这个位置上打字，例如，定义连接分隔符。

> 提示：你应该开启"Add unambiguous imports on the fly"（自动添加明确的导入）选项，便于让IDEA自动添加 `java.util.stream.Collectors` 的导入语句。选项在 `Editor → General → Auto Import` 中。

让我们在实际工作中看看这两个模板：

连接

```
stringCollection
        .stream()
        .filter(s -> s.startsWith("a"))
```

分组

```
stringCollection
        .stream()
        .filter(s -> s.startsWith("a"))
```

Intellij IDEA中的实时模板非常灵活且强大。你可以用它来极大提升代码的生产力。你知道实时模板可以拯救生活的其它例子吗？请让我知道！

仍然不满意吗？在我的数据流教程中学习所有你想要学到的东西。

祝编程愉快！

在 Nashron 中使用 Backbone.js

> 原文：Using Backbone.js with Nashorn
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

这个例子展示了如何在Java8的Nashron JavaScript引擎中使用Backbone.js模型。Nashron在2014年三月首次作为Java SE 8 的一部分发布，并通过以原生方式在JVM上运行脚本扩展了Java的功能。对于Java Web开发者，Nashron尤其实用，因为它可以在Java服务器上复用现有的客户端代码。传统的Node.js具有明显优势，但是Nashorn也能够缩短JVM的差距。

当你在HTML5前端使用现代的JavaScript MVC框架，例如Backbone.js时，越来越多的代码从服务器后端移动到Web前端。这个方法可以极大提升用户体验，因为在使用视图的业务逻辑时节省了服务器的很多往返通信。

Backbone允许你定义模型类，它们可以用于绑定视图（例如HTML表单）。当用户和UI交互时Backbone会跟踪模型的升级，反之亦然。它也能通过和服务器同步模型来帮助你，例如调用服务端REST处理器的适当方法。所以你最终会在前端实现业务逻辑，将你的服务器模型用于处理持久化数据。

在服务端复用Backbone模型十分易于用Nashron完成，就像下面的例子所展示的那样。在我们开始之前，确保你通过阅读我的Nashorn教程熟悉了在Nashron引擎中编写JavaScript。

Java 模型

首先，我们在Java中定义实体类 `Product` 。这个类可用于数据库的CURD操作（增删改查）。要记住这个类是个纯粹的Java Bean，不实现任何业务逻辑，因为我们想让前端正确执行UI的业务逻辑。

```
class Product {
    String name;
    double price;
    int stock;
    double valueOfGoods;
}
```

Backbone 模型

现在我们定义Backbone模型，作为Java Bean的对应。Backbone模型 `Product` 使用和Java Bean相同的数据结构，因为它是我们希望在Java服务器上持久存储的数据。

Backbone模型也实现了业务逻辑： `getValueOfGoods` 方法通过将 `stock` 与 `price` 相乘计算所有产品的总值。每次 `stock` 或 `price` 的变动都会使 `valueOfGoods` 重新计算。

```
var Product = Backbone.Model.extend({
    defaults: {
        name: '',
        stock: 0,
        price: 0.0,
        valueOfGoods: 0.0
    },

    initialize: function() {
        this.on('change:stock change:price', function() {
            var stock = this.get('stock');
            var price = this.get('price');
            var valueOfGoods = this.getValueOfGoods(stock, price
);
            this.set('valueOfGoods', valueOfGoods);
        });
    },

    getValueOfGoods: function(stock, price) {
        return stock * price;
    }
});
```

由于Backbone模型不使用任何Nashron语言扩展，我们可以在客户端（浏览器）和服务端（Java）安全地使用同一份代码。

要记住我特意选择了十分简单的函数来演示我的意图。真实的业务逻辑应该会更复杂。

将二者放在一起

下一个目标是在Nashron中，例如在Java服务器上复用Backbone模型。我们希望完成下面的行为：把所有属性从Java Bean上绑定到Backbone模型上，计算 `valueOfGoods` 属性，最后将结果传回Java。

首先，我们创建一个新的脚本，它仅仅由Nashron执行，所以我们这里可以安全地使用Nashron的扩展。

```
load('http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.6.0/
underscore-min.js');
load('http://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.1.2/ba
ckbone-min.js');
load('product-backbone-model.js');

var calculate = function(javaProduct) {
    var model = new Product();
    model.set('name', javaProduct.name);
    model.set('price', javaProduct.price);
    model.set('stock', javaProduct.stock);
    return model.attributes;
};
```

这个脚本首先加载了相关的外部脚本Underscore 和 Backbone（Underscore是Backbone的必备条件），以及我们前面的 `Product` Backbone模型。

函数 `calcute` 接受 `Product` Java Bean，将其所有属性绑定到新创建的Backbone `Product` 上，之后返回模型的所有属性给调用者。通过在Backbone模型上设置 `stock` 和 `price` 属性， `ValueOfGoods` 属性由于注册在模型 `initialize` 构造函数中的事件处理器，会自动计算出来。

最后，我们在Java中调用 `calculate` 函数。

```
Product product = new Product();
product.setName("Rubber");
product.setPrice(1.99);
product.setStock(1337);

ScriptObjectMirror result = (ScriptObjectMirror)
    invocable.invokeFunction("calculate", product);

System.out.println(result.get("name") + ": " + result.get("value
OfGoods"));
// Rubber: 2660.63
```

我们创建了新的 `Product` Java Bean，并且将它传递到JavaScript函数中。结果触发了 `getValueOfGoods` 方法，所以我们可以从返回的对象中读取 `valueOfGoods` 属性的值。

总结

在Nashron中复用现存的JavaScript库十分简单。Backbone适用于构建复杂的HTML5前端。在我看来，Nashron和JVM现在是Node.js的优秀备选方案，因为你可以在Nashron的代码库中充分利用Java的整个生态系统，例如JDK的全部API，以

及所有可用的库和工具。要记住你在使用Nashron时并不限制于Java -- 想想Scala、Groovy、Clojure和 `jjs` 上的纯JavaScript。

这篇文章中可运行的代码托管在Github上（请见这个文件）。请随意fork我的仓库，或者在Twitter上向我反馈。